MASTER EN SCIENCES INFORMATIQUES

Analyse de strictness des langages applicatifs par l'interprétation abstraite

Pollet, Isabelle

Award date:
1997

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'Informatique

Analyse de "strictness" des langages applicatifs par l'interprétation abstraite

Isabelle Pollet

Mémoire présenté en vue
de l'obtention du grade de
Licencié et Maître en Informatique

Promoteur: B. Le Charlier

Année académique 1996-1997

Résumé

On oppose généralement la sémantique naturelle, d'un point de vue mathématique, du passage par nom à l'efficacité du passage par valeur. L'analyse de "strictness", que l'on situe ici dans le cadre d'un langage applicatif, a pour but la détection des situations indifférentes à l'un ou l'autre type de passage des arguments. Pour soutenir cette analyse, on développe une méthode d'interprétation abstraite basée sur un domaine à deux valeurs. On démontre la correction de cette méthode et on exploite les résultats de l'analyse pour générer un interpréteur "hybride" respectant la définition naturelle du passage par nom mais nettement plus efficace.

Abstract

This work is concerned with strictness analysis in first-order functional language, aimed at detecting situations where calls by name and calls by value yield the same result. This analysis can be used in a "hybrid" interpreter to improve the efficiency: indeed, though calls by name give birth to a more natural semantics (at least from a mathematical point of view), they are greatly less efficient than calls by value. We explain and prove that, this analysis can be realized, in a safe way, by performing abstract interpretation with a two point "definedness" domain.

Avant de commencer ce travail, je tiens tout particulièrement à remercier le professeur Baudouin Le Charlier, mon promoteur. Sans sa gentillesse, ses conseils avisés et surtout sa disponibilité (mêmes aux heures les plus fatidiques), ce mémoire n'aurait certainement jamais vu le jour.

Table des matières

Introduction

L'interprétation abstraite est introduite par beaucoup d'auteurs comme une méthodologie générale de construction d'analyses de programmes. L'idée majeure est de simuler l'exécution réelle d'un processus sur un domaine non-standard, souvent appelé domaine abstrait, pour en dériver des propriétés de ce processus.

Le but de ce travail est de développer complètement un problème par l'interprétation abstraite. Par "complètement", on entend que la démarche doit se préoccuper, d'une part, de décrire un cadre théorique précis et adapté et, d'autre part, de présenter des voies d'implémentation pour les éventuelles solutions.

Le problème considéré se situe dans le cadre des langages applicatifs (i.e. des langages fonctionnels du premier ordre). On y distingue couramment deux types de passages d'arguments : le passage par nom et le passage par valeur. Sans entrer dans une explication complète de ces deux notions, illustrons leurs significations de manière "opérationnelle" sur un exemple.

Considérons la déclaration[1] de fonction suivante :

$$\begin{aligned} f(x,y) = \quad & \textbf{si } x = 0 \\ & \textbf{alors } 1 \\ & \textbf{sinon } f(x-1, f(x,y)). \end{aligned}$$

Le passage par valeur repose sur l'évaluation directe de tous les arguments avant chaque appel de fonction, les paramètres correpondants étant ensuite remplacés par leur valeur dans l'expression de la fonction. On obtient, en appliquant ce principe et en considérant qu'on travaille sur $\mathbb{N}$, les calculs suivants.

[1] On définira évidemment cette notion plus précisément dans la suite de ce travail.

$$
\begin{aligned}
f(0,y) &= 1 \\
f(1,y) &= \textbf{si } 1=0 \\
&\quad \textbf{alors } 1 \\
&\quad \textbf{sinon } f(1-1,f(1,y)) \\
&= f(1-1,f(1,y)) \\
&= f(0.f(\textbf{si } 1=0 \textbf{ alors } 1 \textbf{ sinon } f(1-1,f(1,y)))) \\
&= f(0.f(1-1,f(1,y))) \\
&\ldots \\
&= f(0,f(0,f(0,\ldots f(1-1,f(1,y)))))
\end{aligned}
$$

Dès que le premier argument est strictement positif, on aboutit à une situation de bouclage résultant du fait que l'évaluation de $f(i,y)$ nécessite l'évaluation préalable de $f(i,y)$.

En résumé, la sémantique attendue pour cette déclaration de fonction sera donc

$$
f(x.y) = \begin{cases} 1 & \text{si } x=0 \\ ? & \text{sinon.} \end{cases}
$$

Le passage par nom repose. quant à lui, sur le principe suivant : dans l'expression définissant la fonction, on remplace les occurrences des paramètres par les expressions correspondant aux arguments. On évalue alors l'expression obtenue en évaluant seulement ce qui est nécessaire. Ce principe engendre les développements suivants pour notre exemple.

$$
\begin{aligned}
f(0.y) &= 1 \\
f(1.y) &= \textbf{si } 1=0 \\
&\quad \textbf{alors } 1 \\
&\quad \textbf{sinon } f(1-1,f(1,y)) \\
&= f(1-1,f(1,y))
\end{aligned}
$$

$$
\begin{aligned}
&= \textbf{si } 1-1=0 \\
&\quad \textbf{alors } 1 \\
&\quad \textbf{sinon } f((1-1)-1, f(1-1,y)) \\
\\
&= \textbf{si } 0=0 \\
&= \textbf{alors } 1 \\
&= \textbf{sinon } f((1-1)-1, f(1-1,y)) \\
\\
&= 1
\end{aligned}
$$

On voit que, moyennant des étapes supplémentaires, on obtiendra le même résulat pour n'importe quelle valeur du premier argument. Par conséquent, on peut dire que la sémantique attendue pour f est la fonction constante $\lambda(x,y).1$.

Il se trouve que le passage par nom des arguments correspond à la définition "naturelle", du moins d'un point de vue mathématique[2], du passage des arguments mais s'avère terriblement moins efficace que le passage par valeur des arguments. Ce coût du passage par nom explique d'ailleurs pourquoi la plupart des langages sont implémentés selon une sémantique correspondant au passage par valeur et non au passage par nom.

On peut alors se poser la question : n'y a-t'il pas moyen de détecter les situations où le passage par valeur et le passage par nom aboutissent finalement aux mêmes résultats? Si on parvient à détecter à moindre coût ces situations, on peut alors envisager de remplacer le passage par nom par le passage par valeur dans certains cas. Ce qui permet d'obtenir une méthode intermédiaire qui conserve la définition naturelle du passage par nom mais gagne en efficacité par rapport à cette dernière.

Le principe utilisé, dans ce travail, pour déceler ces situations repose sur l'analyse de "strictness" de Mycroft (cfr [9] et [1]). Ce dernier introduit le concept de fonction stricte. Les fonctions strictes modélisent les fonctions indifférentes à l'un ou l'autre type de passage d'argument.

Afin de pouvoir correctement définir et employer ce concept, on adapte au problème les notions mathématiques générales d'analyse de programmes présentées au cours de *Computational Logic* [3]. Le formalisme et le contexte mathématique utilisés pour l'interprétation abstraite sont quant à eux largement inspirés de l'article [10] .

On développe en fait une modélisation du problème basée sur une formulation dénotationnelle du langage utilisé. Grossièrement, le terme de sémantique dénotationnelle recouvre une formulation mathématique du sens d'un langage par opposition au terme sémantique opérationnelle qui décrit un langage par des règles d'exécution (cfr [8] et [11]).

[2]Selon la théorie que nous allons développer au cours des chapitres suivants.

A partir de ces considérations d'ordre général, on développe une structure en cinq chapitres pour notre exposé.

- Le premier chapitre décrit le cadre mathématique choisi. Il s'agit simplement des définitions des concepts mathématiques employés ainsi que de l'énoncé de quelques propriétés intéressantes de ceux-ci.
- Le deuxième chapitre décrit une sémantique dénotationnelle du langage applicatif étudié. Afin d'illustrer la différence d'efficacité des deux types de passage d'arguments, on fournit en fait une sémantique relative à chaque type de passage d'argument ainsi que des implémentations de celles-ci.
- Le troisième chapitre présente la méthode d'interprétation abstraite utilisée dans un cadre un peu plus général. Il introduit également un concept permettant de traduire la correction la démarche (i.e. le concept d'abstraction sûre d'une fonction) et en dérive des propositions de correction de l'analyse.
- Le quatrième chapitre applique cette méthode au contexte plus délimité de l'analyse de "strictness" et s'attarde particulièrement sur la signification et la construction d'abstractions sûres dans ce contexte restreint.
- Le cinquième chapitre, quant à lui, explique deux algorithmes permettant la réalisation de l'analyse et décrit des implémentations effectives de ceux-ci ainsi que l'insertion de leurs résultats dans un interpréteur "hybride" final.
- La conclusion mentionne les limitations de l'exposé et ses extensions éventuelles.

Signalons, pour boucler cette introduction, que toutes les implémentations sont elles-mêmes réalisées dans un langage de type fonctionnel puisqu'il s'agit du langage CaML (CaML Light version 0.6).

Chapitre 1

Cadre mathématique

Ce chapitre rassemble quelques notions mathématiques courantes qui ne font pas à proprement parler partie du développement qui nous intéresse mais en constituent le soutien indispensable. Le terme "indispensable" ne signifie bien sûr aucunement qu'il s'agisse des seuls concepts pouvant soutenir notre analyse. On entend simplement par là qu'il faut disposer d'outils appropriés pour traduire précisément les objets manipulés, leur sens, leurs propriétés, ...

La sémantique mathématique des langages de programmation repose usuellement soit sur la notion de treillis complet, soit sur celle d'ordre partiel complet. Nous optons quant à nous pour la structure plus faible d'ordre partiel complet.

Ce chapitre présente donc le cadre mathématique relatif à cette structure qui sera la "clé de voûte" du reste du travail. On aborde ainsi successivement les notions d'ordre partiel complet, de domaine primitif, de fonctions continues et de fonctions strictes. On rappelle également une version du théorème du point fixe.

La plupart des propriétés citées dans ce chapitre ne sont pas démontrées ou alors sommairement. Pour des démonstrations complètes, on se réfèrera aux notes du cours de *Computational Logic* [3].

1.1 Notion de CPO

Dans la suite de ce travail, tous les ensembles utilisés seront munis d'une structure d'ordre partiel complet. On utilise plus couramment l'abréviation de la dénomination anglaise "complete partial order" : CPO.

Mais, avant de définir ce qu'est un CPO, il nous faut rappeler quelques définitions préliminaires.

Définition 1.1 (Ordre partiel)

Soit E un ensemble. Le couple $(E, \leq)$ constitue un ordre partiel si et seulement si $\leq$ est une relation réflexive, transitive et antisymétrique sur E.

Définition 1.2 (Minimum)

Soient $(E, \leq)$ un ordre partiel et x, un élément de E. x est un minimum de E si et seulement si pour tout e appartenant à E. $x \leq e$.

La propriété d'antisymétrie entraîne l'unicité d'un tel élément.

Définition 1.3 (Chaîne)

Soit $(E, \leq)$ un ordre partiel. $(x_i)_{i \in I}$ est une chaîne de E si et seulement si

$$\forall i \in I,\ x_i \in E \text{ et } x_i \leq x_{i+1}.$$

Définition 1.4 (Borne supérieure)

Soient $(E, \leq)$ un ordre partiel, x un élement de E et $S \subseteq E$. L 'élément x est une borne supérieure de S si et seulement si

1. $\forall s \in S, s \leq x$,
2. $\forall e \in E, ((\forall s \in S, s \leq e) \Rightarrow e \leq x)$.

Souvent un élément vérifiant la première propriété est appelé "majorant" de S. La borne supérieure est alors définie comme le plus petit des majorants. Une propriété importante réside dans le fait que si la borne supérieure d'un ensemble existe, elle est unique.

On dispose maintenant des notions nécessaires à la définition de notre concept central : l'ordre partiel complet.

Définition 1.5 (Ordre partiel complet[1] : CPO)

Un ordre partiel $(E, \leq)$ est dit complet si et seulement si

[1] Fréquemment, on emploie aussi la dénomination : ensemble inductif.

1. *E admet un minimum*

2. *toute chaîne* $(x_i)_{i \in I}$ *de* E *admet une borne supérieure, notée* $\sqcup_{i \in I} x_i$.

Par la suite, on négligera souvent de mentionner l'index.

Parmi les ordres partiels complets, on en distingue une sous-classe particulière : les domaines primitifs. Les domaines primitifs sont en fait des "constructions de CPO" à partir d'ensembles "plats". Ils sont caractérisés par une relation d'ordre particulière.

Définition 1.6 (Domaine primitif)

Soit un ensemble E. *La construction* $E^+ = E + \{\bot\}$ *munie de la relation d'ordre*

$$x \sqsubseteq y \Leftrightarrow ((x = \bot) \vee (x = y))$$

est appelée domaine primitif sur E.

On prouve facilement que $(E, \sqsubseteq)$ est effectivement un ordre partiel complet dont le minimum est évidemment l'élement ajouté $\bot$. Souvent ce type d'ordre traduit des relations du style "est moins déterminé que". L'élément $\bot$ désigne alors l'indétermination, l'indéfinition.

1.2 Notion de continuité

On ne travaille évidemment pas avec n'importe quelles fonctions. Premièrement, par facilité, on ne conserve que des fonctions totales (autrement appelées applications). Cette restriction ne pose pas de problème, le passage d'une fonction partielle à une application se faisant facilement via des éléments du type "indéterminé".

Une seconde restriction assez courante consiste à ne considérer que des fonctions "continues". Cette notion, très proche de la notion de "calculabilité", permet de disposer d'une version du théorème du point fixe adaptée à la structure de CPO.

On introduit préalablement un concept plus faible, à savoir, le concept de fonction monotone.

Définition 1.7 (Fonction monotone)

Soient $(A, \leq_a)$ *et* $(B, \leq_b)$ *deux ordres partiels, la fonction* $f : A \longrightarrow B$ *est dite monotone si et seulement si*

$$\forall a_1, a_2 \in A, (a_1 \leq_a a_2 \Rightarrow f a_1 \leq_b f a_2).$$

Une fonction monotone est donc une fonction qui respecte la relation d'ordre. Dans le cas des domaines primitifs, si un élément est "moins déterminé" qu'un autre, son image sera nécessairement "moins déterminée" que l'image de l'autre élément. Dans ce sens, il semble naturel de ne conserver que les fonctions monotones : une fonction non monotone peut donner $f\perp > fx$ avec $x \neq \perp$, ce qui signifie que l'image d'un élément non déterminé est strictement plus précise que l'image d'un élément déterminé.

La notion de fonction continue n'a quant à elle de sens que relativement à des structures plus exigeantes telles que celle de CPO.

Définition 1.8 (Fonction continue)

Soient $(A, \leq_a)$ et $(B, \leq_b)$ deux ordres partiels complets, la fonction $f : A \longrightarrow B$ est dite continue si et seulement si elle est monotone et si, pour toute chaîne (a_i) de A, elle vérifie l'égalité

$$\bigsqcup_i^b (fa_i) = f(\bigsqcup_i^a a_i).$$

Remarquons que la monotonie implique déjà un des sens de l'égalité :

$$f \text{ monotone } \Rightarrow \bigsqcup_i^b (fa_i) \leq f(\bigsqcup_i^a a_i).$$

Dans certain cas, la continuité découle directement de la monotonie comme l'indique, par exemple, la propriété ci-dessous.

Proposition 1.1 (Domaine fini)

Soient $(A, \leq_a)$ et $(B, \leq_b)$ deux ordres partiels complets. Si la fonction $f : A \longrightarrow B$ est monotone et si l'ensemble A est fini, alors la fonction f est continue.

La propriété ci-dessous est très souvent implicitement utilisée.

Proposition 1.2 (Composition de fonctions continues)

Soient $(A, \leq_a)$, $(B, \leq_b)$ et $(C, \leq_c)$ des ordres partiels complets.

$$\left.\begin{array}{l} f : A \longrightarrow B \text{ continue} \\ g : B \longrightarrow C \text{ continue} \end{array}\right\} \Rightarrow g \circ f : A \longrightarrow C \text{ continue}$$

1.3 Théorème du point fixe

Nous avons naturellement retenu la version du théorème du point fixe correspondant à la structure mathématique utilisée, à savoir l'ordre partiel complet.

Définition 1.9 (Point fixe)

Soient $f : E \longrightarrow E$ et e appartenant à E. Le point e est un point fixe de la fonction f si et seulement si $fe = e$. Si, pour tout x appartenant à E et tel que $fx = x$, on a $e \leq x$, on parle de plus petit point fixe.

Proposition 1.3 (Théorème du point fixe)

Soit $(E, \leq)$ un ordre partiel complet et une transformation $f : E \longrightarrow E$ continue. Alors, f admet un plus petit point fixe.

La démonstration de ce théorème se base sur la construction de la chaîne :

$$\begin{cases} x_o = \bot \\ x_{i+1} = fx_i. \end{cases}$$

Le plus petit point fixe est alors donné par la borne supérieure $\sqcup_i x_i$. Cette suite est connue sous le nom de "suite de Kleene". Pratiquement, c'est souvent par le calcul des éléments de cette suite qu'on obtient le plus petit point fixe d'une transformation.

Une autre version du théorème du point fixe s'appuie sur la structure de treillis complet. Il s'agit d'une structure plus exigeante que l'ordre partiel complet qui demande l'existence de la borne supérieure de n'importe quelle partie de l'espace. En revanche, cette version ne requiert que la monotonie de la transformation.

1.4 Constructions de CPO

Cette section rassemble quelques propriétés relatives aux procédés de construction de CPO plus "complexes" à partir de structures élémentaires. On s'intéresse ainsi successivement au produit cartésien, à l'union disjointe et à l'ensemble des fonctions continues.

1.4.1 Produit cartésien

Définition 1.10 (Ordre produit)

Soient $(E_i, \leq_i), ..., (E_n, \leq_n)$ *des ordres partiels.*
On définit l'ordre produit $\leq$ *sur* $E_1 \times ... \times E_n$ *par*

$$(e_1, ..., e_n) \leq (e'_1, ..., e'_n) \Leftrightarrow ((e_1 \leq_1 e'_1) \wedge ... \wedge (e_n \leq_n e'_n)).$$

On peut dire en quelque sorte que cet ordre "respecte" la structure de CPO des ensembles élémentaires ce qui s'exprime plus proprement par le théorème suivant.

Proposition 1.4 (CPO produit)

Soient $(E_1, \leq_1), ..., (E_n, \leq_n)$ *des ordres partiels complets. Si on munit* $E = E_1 \times ... \times E_n$ *de l'ordre produit* $\leq$*, alors* $(E, \leq)$ *est également un ordre partiel complet.*

La preuve de cette propriété s'appuie sur la construction de l'élément minimum $(\bot_{E_1}, ..., \bot_{E_n})$ et sur la "définition" de la borne supérieure produit

$$\bigsqcup_i e^i = \bigsqcup_i (e^i_1, ..., e^i_n) = (\bigsqcup_i^1 e^i_1, ..., \bigsqcup_i^n e^i_n).$$

On dispose également d'une proposition "d'extension" de la continuité de fonctions.

Proposition 1.5 (Fonctions continues pour des espaces produits)

Soient $(A_1, \leq^a_1), ..., (A_n, \leq^a_n)$ *et* $(B_1, \leq^b_1), ..., (B_n, \leq^b_n)$ *des ordres partiels complets. Si les fonctions* $f_1 : A_1 \longrightarrow B_1, ..., f_n : A_n \longrightarrow B_n$ *sont continues alors la fonction* f *définie par*

$$\begin{array}{rccc} f: & A_1 \times ... \times A_n & \longrightarrow & B_1 \times ... \times B_n \\ & (a_1, ..., a_n) & \rightsquigarrow & (f_1 a_1, ..., f_n a_n) \end{array}$$

est continue pour les ordres produits.

Cette propriété pourrait s'énoncer sous la forme "le produit de fonctions continues est continu".

1.4.2 Union disjointe

On peut suivre un schéma tout à fait analogue à celui suivi pour le produit cartésien si on définit notre union disjointe de $E_1, ..., E_n$ par

$$E_1 + ... + E_n = \{\bot\} \cup \{(i, e_i) \mid i \in \{1, ..., n\} \wedge e_i \in E_i\}.$$

Définition 1.11 (Ordre somme)

Soient $(E_1, \leq_1), ..., (E_n, \leq_n)$ *des ordres partiels.*
On définit l'ordre somme $\leq$ *sur* $E = E_1 + ... + E_n$ *par*

$$e \leq e' \Leftrightarrow (e = \bot) \vee ((e = (i, e_i)) \wedge (e' = (i, e_i')) \wedge (e_i \leq_i e_i')).$$

La proposition suivante exprime que cette construction "respecte" la structure de CPO.

Proposition 1.6 (CPO somme)

Soient $(E_1, \leq_1), ..., (E_n, \leq_n)$ *des ordres partiels complets. Si on munit* $E = E_1 + ... + E_n$ *de l'ordre somme* $\leq$*, alors* $(E, \leq)$ *est également un ordre partiel complet.*

On a évidemment que $\bot$ joue le rôle du minimum de E. On construit la borne supérieure d'un chaîne (e_i) de E selon la démarche :

- si tous les e_i valent $\bot$, la borne supérieure vaut également $\bot$,
- sinon, c'est-à-dire si un élément de la suite est de la forme $e_k = (j, e_k^j)$ avec e_k^j appartenant à E_j, la borne supérieure s'écrit

$$\bigsqcup_i e_i = \bigsqcup_i (j, e_i^j) = (j, \bigsqcup_i^j e_i^j).$$

Pour passer facilement de la structure composite aux structures élémentaires et réciproquement, on peut construire des opérations de projection et d'injection.

$$\begin{array}{lccll} p_{E_i} : & E_1 + ... + E_n & \longrightarrow & E_i & \\ & \bot & \rightsquigarrow & \bot_{E_i} & \\ & (j, e^j) & \rightsquigarrow & \bot_{E_i} & \text{si } i \neq j \\ & (i, e^i) & \rightsquigarrow & e^i & \end{array}$$

$$\begin{array}{lccl} in_{E_i} : & E_i & \longrightarrow & E_1 + ... + E_n \\ & e^i & \rightsquigarrow & (i, e^i) \end{array}$$

Proposition 1.7 (Continuité des applications de projection et d'injection)

Si $(E_1, \leq_1), ..., (E_n, \leq_n)$ *et* $(E, \leq)$ *sont des ordres partiels complets avec* $E = E_1 + ... + E_n$ *et* $\leq$ *représentant l'ordre somme, les applications* p_{E_i} *et* in_{E_i} *sont continues.*

Cette propriété découle directement de la définition des applications concernées et de la construction de la borne supérieure dans E.

On dispose également d'une proposition "d'extension" de la continuité de fonctions.

Proposition 1.8 (Fonctions continues pour des espaces sommes)

Soient $(A_1, \leq_1^a), ..., (A_n, \leq_n^a)$ *et* $(B_1, \leq_1^b), ..., (B_n, \leq_n^b)$ *des ordres partiels complets. Si les fonctions* $f_1 : A_1 \longrightarrow B_1, ..., f_n : A_n \longrightarrow B_n$ *sont continues alors la fonction* f *définie par*

$$\begin{array}{rccc} f: & A_1 + ... + A_n & \longrightarrow & B_1 + ... + B_n \\ & \perp_a & \rightsquigarrow & \perp_b \\ & (j, a^j) & \rightsquigarrow & (j, f_j a^j) \end{array}$$

est continue pour les ordres sommes.

Cette propriété pourrait s'énoncer sous la forme "la somme de fonctions continues est continue".

1.4.3 Ensemble des fonctions continues

Définition 1.12 (Ordre "point à point" pour les fonctions)

Soit $(E, \leq_E)$ *un ordre partiel et* D *un ensemble. On définit l'ordre "point à point"* $\leq$ *sur* $D \longrightarrow E$ *par*

$$f \leq g \Leftrightarrow (\forall d \in D, f d \leq_E g\ d).$$

A partir de ce nouvel ordre, on peut construire un nouvel ordre partiel complet sur l'ensemble des fonctions.

Proposition 1.9 (CPO ensemble des fonctions continues)

Soient $(E_1, \leq_1)$ *et* $(E_2, \leq_2)$ *deux ordres partiels complets. L'ensemble des fonctions continues de* E_1 *dans* E_2 *muni de l'ordre "point à point"* E_2 *est également un ordre partiel complet.*

La preuve de cette propriété se décompose comme suit :

- on montre que $\lambda d.\bot_{E_2}$ est l'élément minimum de $E_1 \longrightarrow E_2$ et qu'il s'agit bien d'une fonction continue,
- on construit la borne supérieure $\sqcup_i f_i$ d'une chaîne (f_i) de $E_1 \longrightarrow E_2$ "point à point"

$$\forall e_1 \in E_1,\ (\bigsqcup_i f_i)e_1 = \bigsqcup_i{}^2 (f_i e_1),$$

- on montre que si toutes les fonctions f_i sont continues alors la borne supérieure $\sqcup_i f_i$ est également continue (il s'agit d'ailleurs du passage délicat de la preuve).

Par la suite, on symbolisera souvent l'ensemble des fonctions continues de E_1 dans E_2 par $[E_1 \longrightarrow E_2]$.

On utilise souvent ce théorème sous une forme simplifiée qui exprime simplement qu'on peut "reporter" la structure de CPO d'un ensemble image E sur l'ensemble des fonctions $D \longrightarrow E$.

Proposition 1.10 (CPO ensemble des fonctions)

Soit $(E, \leq_E)$ un ordre partiel complet et D un ensemble quelconque. Si on munit $D \longrightarrow E$ de l'ordre "point à point", on obtient une nouvelle structure de CPO.

Pour passer de la proposition 1.9 à celle-ci, il suffit de considérer l'ordre trivial sur l'ensemble D

$$d_1 \leq_D d_2 \Leftrightarrow d_1 = d_2.$$

On a alors que chaque fonction f de $D \longrightarrow E$ est continue puisque les seules chaînes de $(D, \leq_D)$ sont les suites stationnaires. On retombe alors sur la proposition précédente.

1.5 Fonctions strictes

Un de nos objectifs est de trouver une expression dénotationnelle de la sémantique d'un programme dans un langage applicatif et ce d'une part pour le passage valeur des arguments et d'autre part pour le passage par nom.

On part de l'idée qu'un programme détermine une transformation des sémantiques des fonctions. La sémantique d'un programme est alors définie comme le plus petit point fixe de la transformation déterminée par ce même programme. Le chapitre suivant a d'ailleurs pour but de formuler précisément cette idée.

1.5.1 Illustration

Illustrons ce principe, de manière tout à fait informelle, sur le petit exemple présenté au cours de l'introduction.

On considère donc le programme composé de la seule déclaration

$$\begin{array}{ll} f(x,y) = & \textbf{si } x = 0 \\ & \textbf{alors } 1 \\ & \textbf{sinon } f(x-1, f(x,y)). \end{array}$$

Comme précisé dans l'introduction, on travaille dans l'ensemble $\mathbb{N}$. On adjoint à cet ensemble un élément $\perp$ symbolisant l'indétermination. La sémantique mathématique de f est modélisée par une fonction continue de $\mathbb{N}^+ \times \mathbb{N}^+ \longrightarrow \mathbb{N}^+$.

La déclaration de la fonction détermine une transformation $\mathcal{D}$ de l'espace $[\mathbb{N}^+ \times \mathbb{N}^+ \longrightarrow \mathbb{N}^+]$ qui vérifie

$$\begin{array}{ll} \mathcal{D}\ g = \lambda(x,y). & \textbf{si } x = 0 \\ & \textbf{alors } 1 \\ & \textbf{sinon } g(x-1, g(x,y)) \end{array}$$

Calculons maintenant le plus petit point fixe de cette transformation[2]. Pour ce faire, on construit pas à pas la suite de Kleene correspondante.

$$\begin{array}{lll} f_0 & = \lambda(x,y). & \perp \\ f_1 & = \mathcal{D}\ f_0 & \\ & = \lambda(x,y). & \textbf{si } x = 0 \\ & & \textbf{alors } 1 \\ & & \textbf{sinon } f_0(x-1, f_0(x.y)) \\ & = \lambda(x,y). & \textbf{si } x = 0 \\ & & \textbf{alors } 1 \\ & & \textbf{sinon } \perp \end{array}$$

[2]On admettra la continuité de la transformation $\mathcal{D}$, une démonstration plus générale étant fournie au chapitre suivant.

$$
\begin{aligned}
f_2 &= \mathcal{D}f_1 \\
&= \lambda(x,y).\ \begin{array}{l}\textbf{si } x = 0 \\ \textbf{alors } 1 \\ \textbf{sinon } f_1(x-1, f_1(x,y))\end{array} \\
&= \lambda(x,y).\ \begin{array}{l}\textbf{si } x = 0 \\ \textbf{alors } 1 \\ \quad\textbf{sinon}\quad \begin{array}{l}\textbf{si } x - 1 = 0 \\ \textbf{alors } 1 \\ \textbf{sinon } \bot\end{array}\end{array} \\
&= \lambda(x,y).\ \begin{array}{l}\textbf{si } x < 2 \\ \textbf{alors } 1 \\ \textbf{sinon } \bot\end{array}
\end{aligned}
$$

On peut prouver par récurrence que

$$
\begin{aligned}
f_i &= \mathcal{D}f_{i-1} \\
&= \lambda(x,y).\ \begin{array}{l}\textbf{si } x = 0 \\ \textbf{alors } 1 \\ \textbf{sinon } f_{i-1}(x-1, f_{i-1}(x,y))\end{array} \\
&= \lambda(x,y).\ \begin{array}{l}\textbf{si } x < i \\ \textbf{alors } 1 \\ \textbf{sinon } \bot.\end{array}
\end{aligned}
$$

On peut maintenant calculer le plus petit point fixe, comme la borne supérieure de suite de Kleene venant d'être définie.

L'expression des fonctions f_i entraîne évidemment que pour tout point (x, y), il existe un i suffisamment grand pour que $f_i(x, y) = 1$. Par conséquent, on calcule aisément la borne supérieure désirée point par point.

$$
\begin{aligned}
\bigsqcup_i f_i &= \lambda(x,y).\bigsqcup_i (f_i(x,y)) \\
&= \lambda(x,y).1
\end{aligned}
$$

Cet exemple illustre la coïncidence de l'expression dénotationnelle de la sémantique d'un programme par le plus petit point fixe d'une transformation et de la sémantique

opérationnelle basée sur le passage par nom des arguments (sémantique décrite dans l'introduction).

Comment à présent obtenir une sémantique analogue pour le passage par valeur? Pour répondre à cette question, on se pose la question adjacente: quand le passage par valeur et le passage par nom fournissent-ils des résultats différents?

Dans le cadre des langages fonctionnels, les seuls cas de divergence résultent d'appels de fonctions n'utilisant pas "réellement" un argument dont le calcul conduirait à une exécution infinie. Pour identifier ces fonctions, on introduit le concept de "fonction stricte". Intuitivement, une fonction sera stricte en un argument si l'indétermination de celui-ci entraîne l'indétermination de la valeur de la fonction.

Ainsi, dans notre exemple, la sémantique mathématique de f basée sur la notion de plus petit point fixe est stricte en la première variable mais pas en la seconde.

Une sémantique opérationnelle basée sur l'appel par valeur implique que toutes les fonctions sont strictes en tous leurs arguments.

1.5.2 Définitions

On formalise ici le concept de fonction stricte introduit dans l'exemple de la section précédente.

Définition 1.13 (Fonction stricte: cas à une dimension)

Soient $(A, \leq_a)$ *et* $(B, \leq_b)$ *deux CPO et soit* $f : A \longrightarrow B$ *une fonction continue. On dira que* f *est une fonction stricte si et seulement si* $f\perp_A = \perp_B$.

Intuitivement, cette définition signifie bien que l'application d'une fonction stricte à un élément indéterminé ne peut fournir qu'un résultat indéterminé.

Remarquons que les seules fonctions à une variable qui n'utilisent pas leur unique argument sont les fonctions constantes et que la seule fonction constante stricte est la fonction toujours indéterminée. Pour cette dernière, l'appel par nom et l'appel par valeur fourniront évidemment le même résultat tandis que pour toutes les autres fonctions constantes, si l'argument est indéterminé, les deux sémantiques diffèreront. Le concept introduit couvre donc bien les cas désirés.

Définition 1.14 (Fonction stricte en un argument)

Soient $(A_1, \leq_{a_1})$, ..., $(A_n, \leq_{a_n})$ *et* $(B, \leq_b)$ *des CPO et soit* $f : A_1 \times ... \times A_n \longrightarrow B$ *continue. On dira que* f *est stricte en son i-ème argument si et seulement si*

$$\forall (a_1..., a_{i-1}, a_{i+1}, ...a_n) \in A_1 \times ... \times A_{i-1} \times A_{i+1} \times ... \times A_n,$$
$$(a_1, ..., a_{i-1}, \perp_{A_i}, a_{i+1}, ..., a_n) = \perp_B$$

On dira qu'une fonction est stricte si elle stricte en tous ses arguments. D'une fonction non stricte, on dira parfois qu'elle est "lazy".

Chapitre 2

Description d'un langage applicatif

Dans un premier temps, ce chapitre se consacre à la présentation et à la définition du langage étudié, à savoir un langage applicatif. Il se doit donc de fournir une syntaxe abstraite agrémentée de quelques règles additionnelles et une sémantique.

On présente en fait deux sémantiques : la première adaptée au passage par nom des arguments et la seconde au passage par valeur. Toutes deux revètent une forme dénotationnelle et s'appuient sur les concepts mathématiques présentés dans le chapitre précédent : la structure d'ordre partiel complet et le théorème du point fixe.

Dans un second temps, ce chapitre présente des implémentations pour chacune des deux sémantiques mentionnées ci-dessus. Ces implémentations sont elle-mêmes réalisées dans un langage fonctionnel : CaML.

Ces implémentations permettent finalement via quelques exemples d'illustrer l'efficacité relative du passage par valeur sur le passage par nom.

2.1 Syntaxe abstraite

La syntaxe abstraite d'un langage applicatif du premier ordre est assez simple : il s'agit juste de construire des expressions et des déclarations de fonctions à partir d'ensembles élémentaires de variables, de symboles de base, de symboles fonctionnels et de fonctions prédéfinies.

Pour désigner ces ensembles élémentaires, on emploie les notations suivantes :

- $\mathbb{V}ar$ représente l'ensemble des variables,
- $\mathbb{B}ase$ représente l'ensemble des symboles de base (également appelés constantes),

- $\mathbb{F}onc$ représente l'ensemble des symboles fonctionnels,
- $\mathbb{P}rim$ représente l'ensemble des symboles fonctionnels prédéfinis, (également appelées primitives).

A partir de ces derniers, on construit les autres ensembles d'intérêt, à savoir :

- $\mathbb{E}xpr$, l'ensemble des expressions,
- $\mathbb{D}ec$, l'ensemble des déclarations,
- $\mathbb{P}rog$, l'ensemble des programmes,
- $\mathbb{E}xec$, l'ensemble des "exécutions".

Le procédé de construction est fourni par la syntaxe abstraite ci-dessous.

$$
\begin{array}{l}
x \in \mathbb{V}ar \\
f \in \mathbb{F}onc \\
b \in \mathbb{B}ase \\
prim \in \mathbb{P}rim \\
expr \in \mathbb{E}xpr \\
dec \in \mathbb{D}ec \\
prog \in \mathbb{P}rog \\
exec \in \mathbb{E}xec
\end{array}
$$

$$
\begin{array}{lcl}
expr & ::= & b \mid x \mid prim\ expr^* \mid f\ expr^* \\
dec & ::= & f\ x^* = expr \\
prog & ::= & \textbf{vide} \mid dec\ prog \\
exec & ::= & \textbf{let}\ prog\ \textbf{in}\ expr^*
\end{array}
$$

La notation $expr^*$ désigne une séquence, peut-être vide, d'expressions.

Une expression est donc soit un symbole de base, soit une variable, soit une primitive "appliquée" à une séquence d'expressions, soit encore un symbole de fonction et une séquence d'expressions. Une déclaration est composée d'un symbole fonctionnel, d'une séquence de variables et d'une expression. Un programme est une séquence de déclarations de fonctions. Une "exécution" est composée d'un programme et d'expressions.

2.2 Règles syntaxiques additionnelles

Il est évident qu'on ne peut pas admettre n'importe quelle "exécution" vérifiant la

syntaxe abstraite présentée dans la section précédente. On adjoint donc à cette dernière quelques règles usuelles qui restreignent les ensembles construits.

Cette section rassemble les énoncés informels de ces règles ainsi qu'une formulation "ensembliste" de celles-ci.

2.2.1 Quelques définitions préliminaires

Une première définition concerne les primitives: il va de soi que chaque primitive dispose d'une arité unique prédéfinie.

Définition 2.1 ($\mathcal{A}r$)

$\mathcal{A}r$: $\mathbb{P}rim \longrightarrow \mathbb{N}$ fournit l'arité d'une primitive donnée.

Les définitions qui suivent se font toutes "par induction sur la structure" des éléments concernés.

Définition 2.2 ($\mathcal{V}ars$)

$\mathcal{V}ars$: $\mathbb{E}xpr \longrightarrow \wp(\mathbb{V}ar)$ fournit l'ensemble des variables ayant au moins une occurence dans une expression donnée.

$$\begin{aligned}
&\mathcal{V}ars(b) = \phi \\
&\mathcal{V}ars(x) = \{x\} \\
&\mathcal{V}ars(prim\ expr_1, ..., expr_n) = \bigcup_{i=1}^{n} \mathcal{V}ars(expr_i) \\
&\mathcal{V}ars(f\ expr_1, ..., expr_n) = \bigcup_{i=1}^{n} \mathcal{V}ars(expr_i)
\end{aligned}$$

Par souci de simplicité, on utilise abusivement la notation $expr_1, ..., expr_n$ pour une séquence $expr^*$; la séquence vide correspondant au cas $n = 0$.

Définition 2.3 ($\mathcal{F}oncts$)

$\mathcal{F}oncts$: $\mathbb{E}xpr \longrightarrow \wp(\mathbb{F}onc)$ fournit l'ensemble des symboles fonctionnels ayant au moins une occurence dans une expression donnée.

$$\begin{aligned}
&\mathcal{F}oncts(b) = \phi \\
&\mathcal{F}oncts(x) = \phi \\
&\mathcal{F}oncts(prim\ expr_1, ..., expr_n) = \bigcup_{i=1}^{n} \mathcal{F}oncts(expr_i) \\
&\mathcal{F}oncts(f\ expr_1, ..., expr_n) = (\bigcup_{i=1}^{n} \mathcal{F}oncts(expr_i)) \cup \{f\}
\end{aligned}$$

Définition 2.4 ($\mathcal{V}erifar$)

$\mathcal{V}erifar : \mathbb{E}xpr \longrightarrow \mathbb{B}ool$ exprime si une expression respecte les arités des primitives.

$\mathcal{V}erifar(b) = \underline{vrai}$
$\mathcal{V}erifar(x) = \underline{vrai}$
$\mathcal{V}erifar(prim\ expr_1, ..., expr_n) = (Ar(prim) = n)\ \underline{et}\ (\forall i \in \{1, ..., n\},\ \mathcal{V}erifar(expr_i))$
$\mathcal{V}erifar(f\ expr_1, ..., expr_n) = (\forall i \in \{1, ..., n\},\ \mathcal{V}erifar(expr_i))$

Définition 2.5 ($\mathcal{A}rite$)

$\mathcal{A}rite : \mathbb{F}onc \times \mathbb{P}rog \longrightarrow \mathbb{N} \cup \{?\}$ fournit l'arité de la première occurence du symbole fonctionnel rencontré dans le programme[1].

$\mathcal{A}rite(f, \textbf{vide}) = ?$
$\mathcal{A}rite(f, f x_1, ..., x_n = expr\ prog) = n$
$\mathcal{A}rite(f, g\ x^* = expr\ prog) = \mathcal{A}rite(f, prog)$

Définition 2.6 ($\mathcal{V}erifarite$)

$\mathcal{V}erifarite : \mathbb{P}rog \times \mathbb{E}xpr \longrightarrow \mathbb{B}ool$ teste si les appels de fonctions d'une expression respectent les arités déclarées dans un programme.

$\mathcal{V}erifarite(p, b) = \underline{vrai}$
$\mathcal{V}erifarite(p, x) = \underline{vrai}$
$\mathcal{V}erifarite(p, prim\ expr_1, ..., expr_n) = (\forall i \in \{1, ..., n\},\ \mathcal{V}erifarite(p, expr_i))$
$\mathcal{V}erifarite(p, f\ expr_1, ..., expr_n)$
$\quad = (\forall i \in \{1, ..., n\},\ \mathcal{V}erifarite(p, expr_i))\ \underline{et}\ (\mathcal{A}rite(f, p) = n)$

2.2.2 Expressions

La seule règle ajoutée pour les expressions concerne les arités des primitives qui doivent évidemment être respectées. L'arité des fonctions n'a, quant à elle, de sens que relativement à un programme donné.

$$\frac{expr \in \mathbb{E}xpr \quad \mathcal{V}erifar(expr)}{\vdash expr}$$

La notation "$\vdash expr$" signifie que $expr$ est une expression "acceptable".

[1] Rien ne garantit ici qu'un symbole fonctionnel intervienne dans l'en-tête d'une seule déclaration.

2.2.3 Déclarations

Une déclaration est acceptable si les seules variables qui apparaissent dans son expression sont les variables mentionnées dans son en-tête. Ces dernières variables doivent toutes être distinctes.

$$\frac{\begin{array}{l}\vdash expr \\ \forall (i,j) \in \{1,...,n\}^2,\ i \neq j \Rightarrow x_i \neq x_j \\ \mathcal{V}ars(expr) \subseteq \{x_1,...,x_n\}\end{array}}{\vdash f(x_1,...,x_n) = expr}$$

2.2.4 Programmes

Un programme est acceptable si

- une fonction n'est déclarée qu'une seule fois,
- les symboles fonctionnels mentionnés dans les expressions correspondent à des fonctions déclarées,
- les appels de fonctions apparaissant dans les expressions respectent les arités déclarées dans le programme.

$$\frac{}{\vdash \textbf{vide}}$$

$$\frac{\begin{array}{l}\forall i\ (\vdash f_i x_i^* = expr_i) \\ (\forall (i,j) \in \{1,...,n\}^2,\ i \neq j \Rightarrow f_i \neq f_j \\ \bigcup_{i=1}^{n} \mathcal{F}oncts(expr_i) \subseteq \{f_1,...,f_n\} \\ \forall i \in \{1,....n\},\ \mathcal{V}erifarite((f_1 x_1^* = expr_1,...,f_n x_n^* = expr_n), expr_i)\end{array}}{\vdash (f_1 x_1^* = expr_1,...,f_n x_n^* = expr_n)}$$

2.2.5 Exécutions

Une exécution est acceptable si

- le programme mentionné est acceptable,
- les appels de fonctions apparaissant dans les expressions correspondent aux déclarations du programme,

- les expressions ne contiennent pas de variables.

$$\frac{\begin{array}{l}\vdash prog \\ \forall i \in \{1,...,n\},\ (\vdash expr_i) \\ \forall i \in \{1,...,n\},\ (\mathcal{V}ars(expr_i) = \phi) \\ \forall i \in \{1,...,n\},\ (\mathcal{V}erifarite(prog, expr_i))\end{array}}{\vdash \textbf{let}\ prog\ \textbf{in}\ expr_1, ..., expr_n}$$

2.3 Sémantique pour le passage par nom

Cette section fournit successivement une sémantique pour les constantes, les primitives, les expressions, les programmes et les exécutions. On y manipule, en fait, les ensembles restreints obtenus à partir des règles additionnnelles présentées dans la section précédente. Ainsi, lorsque l'on cite l'ensemble $\mathbb{E}xpr$, on se réfère implicitement à l'ensemble

$$\mathbb{E}xpr' = \{expr \in \mathbb{E}xpr \mid \vdash expr\}.$$

Les principaux "outils" utilisés sont la structure de CPO et le théorème du point fixe s'appuyant sur cette dernière.

2.3.1 Sémantique des constantes

Chaque constante est simplement interprété comme un élément d'un domaine D, appelé domaine de valeurs. Cet ensemble muni d'une relation d'ordre notée $\sqsubseteq$ possède une structure de CPO. Son élément minimum est symbolisé par $\perp$.

Effectivement, c'est-à-dire dans les cas instanciés du langage (constantes et primitives particulières), ce domaine prend la forme plus précise d'un domaine primitif ou d'une construction de domaines primitifs. L'élément minimum revêt alors son sens habituel d'indétermination ou de non-terminaison.

L'interprétation d'un élément b de $\mathbb{B}ase$ s'écrit $\tilde{b}$.

2.3.2 Sémantique des primitives

Intuitivement, on désire qu'une primitive d'arité n soit interprêtée par une application prédéfinie de $[D^n \longrightarrow D]$ (on ne considère que les applications continues).

Pour obtenir un traitement indépendant du nombre d'arguments (i.e. de l'arité), on définit

$$D^* = \bigoplus_{i \in \mathbb{N}} D^i = \{(n, d) \mid n \in \mathbb{N}, d \in D^n\}$$

On peut facilement passer de D^* à D^i et réciproquement par des applications de projection p^i et d'injection in^i.

On interprète alors chaque primitive *prim* comme un élément de $[D^* \longrightarrow D]$ noté $\widetilde{prim}$, les seuls éléments d'intérêt étant de la forme $\widetilde{prim} = prim^n \circ p^n$ avec $prim^n \in [D^n \longrightarrow D]$.

Comme on l'a vu au chapitre précédent, on peut facilement reporter la structure de CPO d'un domaine simple sur les constructions habituelles de domaines à partir de celui-ci. On obtient ainsi successivement.

- $\forall i \geq 1$, $(D^i, \sqsubseteq^i)$ possède une structure de CPO pour l'ordre produit usuel (cfr proposition 1.4).

- En munissant D^* de l'ordre somme (noté $\sqsubseteq^*$), on obtient de nouveau une structure de CPO (cfr proposition 1.6).

- Comme D^* et D sont, tous deux, des CPO on obtient que $[D^* \longrightarrow D] \stackrel{not}{=} D^>$ muni de l'ordre "point-à-point" habituel (notation : $\sqsubseteq^>$) est aussi un CPO (cfr proposition 1.9).

On utilise par la suite les notations suivantes : $\perp^i$ représente l'élément minimum de D^i tandis qu'une borne supérieure dans cet espace est notée $\sqcup^i$; de façons analogues, on introduit les symboles $\perp^*$ et $\sqcup^*$ pour l'espace D^* et les symboles $\perp^>$ et $\sqcup^>$ pour l'espace $D^>$.

$$\begin{cases} \perp^2 & = (\perp, \perp) \\ \perp^* & = (0, d^0) \text{où } d^0 \text{ représente le seul élément de } D^0 \\ \perp^> & = \lambda d^*.\perp \end{cases}$$

On peut maintenant expliciter les applications de projection et d'injection mentionnées précédemment.

$$
\begin{array}{lccll}
p^i : & D^* & \longrightarrow & D^i & \\
 & \perp^* & \rightsquigarrow & \perp^i & \\
 & (j, d^j) & \rightsquigarrow & \perp^i & \text{si } i \neq j \\
 & (i, d^i) & \rightsquigarrow & d^i & \\
 & & & & \\
in^i : & D^i & \longrightarrow & D^* & \\
 & d^i & \rightsquigarrow & (i, d^i) &
\end{array}
$$

Par la proposition 1.7, on sait en outre qu'il s'agit d'applications continues pour les ordres $\sqsubseteq^*$ et $\sqsubseteq^i$.

2.3.3 Sémantique des expressions

Une expression n'a de sens que si les éléments qui la composent ont un sens ... Nous avons déjà précisé la signification des constantes et des primitives. Restent les variables et les symboles fonctionnels.

La sémantique d'une variable doit être un élément du domaine de valeurs et celle d'un symbole fonctionnel une application de $[D^* \longrightarrow D]$ mais ces significations ne peuvent être "absolues". La sémantique d'une variable dépend de l'appel de fonction en cours d'évaluation et la sémantique d'une fonction est déterminée par un programme donné.

L'ensemble de ces sémantiques (i.e. les sémantiques de tous les éléments de $\mathbb{V}ar$ et de $\mathbb{F}onc$) décrit en quelque sorte "les conditions d'évaluation d'une expression". Pour désigner, cet "ensemble", on emploie souvent le terme environnement. On choisit ici de distinguer un "environnement pour les variables" et un "environnement pour les fonctions".

Définition 2.7 ($\mathbb{E}nvVar$)

$\mathbb{E}nvVar = \mathbb{V}ar \longrightarrow D$ est l'ensemble des environnements pour les variables.

On peut évidemment reporter la structure de CPO de D sur $\mathbb{E}nvVar$ (cfr proposition 1.10). On note l'élément minimum $\mathcal{V}_\perp$.

Définition 2.8 ($\mathbb{E}nvFonc$)

$\mathbb{E}nvFonc = \mathbb{F}onc \longrightarrow [D^* \longrightarrow D]$ est l'ensemble des environnements pour les symboles fonctionnels.

De nouveau, on peut reporter la structure de CPO de $[D^* \longrightarrow D]$ sur *IEnvFonc*.

On peut maintenant définir une fonction d'évaluation des expressions, par induction sur la longueur des expressions.

Définition 2.9 (Fonction d'évaluation : $\varepsilon[\![.]\!]^2$)

ε : *IExpr* $\longrightarrow$ *IEnvFonc* $\longrightarrow$ *IEnvVar* $\longrightarrow$ D *fournit la valeur d'une expression étant donnés un environnement de fonctions et un environnement de variables.*

Soient $\mathcal{V} \in$ *IEnvVar* et $\mathcal{F} \in$ *IEnvFonc*.

$$\varepsilon[\![b]\!]\mathcal{V}\mathcal{F} = \tilde{b}$$
$$\varepsilon[\![x]\!]\mathcal{V}\mathcal{F} = \mathcal{V}(x)$$
$$\varepsilon[\![prim\ expr_1, ..., expr_n]\!]\mathcal{V}\mathcal{F} = \widetilde{prim}(in^n(\varepsilon[\![expr_1]\!]\mathcal{V}\mathcal{F}, ..., \varepsilon[\![expr_n]\!]\mathcal{V}\mathcal{F}))$$
$$\varepsilon[\![f\ expr_1, ..., expr_n]\!]\mathcal{V}\mathcal{F} = (\mathcal{F}f)(in^n(\varepsilon[\![expr_1]\!]\mathcal{V}\mathcal{F}, ..., \varepsilon[\![expr_n]\!]\mathcal{V}\mathcal{F}))$$

2.3.4 Sémantique des programmes

L'effet d'un programme doit être de donner un sens aux différentes fonctions déclarées dans celui-ci. La sémantique d'un programme est donc en fait un environnement de fonctions particulier. Il faut trouver un environnement de fonctions qui vérifie toutes les "équations du programme" c'est-à-dire $\mathcal{F} \in$ *IEnvFonc* tel que pour toute déclaration $fx_1, ..., x_n = expr$ du programme

$$(\mathcal{F}f(in^n(\mathcal{V}(x_1), ..., \mathcal{V}(x_n))) = \varepsilon[\![expr]\!]\mathcal{V}\mathcal{F}$$

On prendra le "moins exigeant de ces environnements", c'est-à dire le plus petit. s'il existe.

Cette démarche revient à prendre le plus petit point fixe d'une transformation de *IEnvFonc* qui décrirait "l'effet d'un programme" sur un environnement de fonctions donné.

On définit cette transformation récursivement sur la longueur du programme. Un programme vide ne doit engendrer aucune modification de l'environnement de fonctions. L'ajout d'une déclaration à un programme modifie l'environnement obtenu pour ce programme en un seul point : le symbole fonctionnel de la déclaration.

[2] On utilise la notation usuelle [.] pour encadrer les éléments de type syntaxique. Cependant, pour ne pas alourdir trop certaines formules, on ne respectera pas cette règle pour les variables et les symboles fonctionnels.

Définition 2.10 (Transformation d'un environnement : $\tau[\![p]\!]$)

τ : IProg $\longrightarrow$ IEnvFonc $\longrightarrow$ IEnvFonc fournit la transformation "exigée" par un programme.

Soit $\mathcal{F} \in$ *IEnvFonc*.

$$\tau[\![\textbf{vide}]\!]\mathcal{F} = \mathcal{F}$$
$$\tau[\![f\ (x_1, ..., x_n) = expr\ prog]\!]\mathcal{F} = (\tau[\![prog]\!]\mathcal{F})[f/\lambda v^*.\varepsilon[\![expr]\!]\mathcal{F}\mathcal{V}_{v^*}]$$

où la notation $\mathcal{F}[f/F]$ symbolise, comme de coutume, la fonction égale à $\mathcal{F}$ en tous points sauf en f où elle prend la valeur F et où $\mathcal{V}_{v^*}$ est défini par

$$\mathcal{V}_{v^*} = \mathcal{V}_\perp[x_1/v_1, ..., x_n/v_n] \text{ avec } p^n(v^*) = (v_1, ..., v_n)$$

Définition 2.11 (Sémantique d'un programme : $\mu[\![prog]\!]$)

μ : IProg $\longrightarrow$ IEnvFonc fournit le plus petit point fixe de la transformation $\tau[\![prog]\!]$, dénommé "sémantique de p".

Cette définition n'a évidemment de sens que si on peut garantir l'existence de ce point fixe. Ce qui sera effectivement le cas si *IEnvFonc* est un CPO et si pour tout p appartenant à *IProg*, la transformation $\tau[\![p]\!]$ est continue par rapport à cette structure (cfr théorème du point fixe). Il nous faut donc prouver la continuité de $\tau[\![p]\!]$.

Pour ce faire, on commence par prouver la continuité de la fonction sémantique d'évaluation des expressions par rapport aux environnements de fonctions.

Proposition 2.1 (Continuité de $\varepsilon[\![expr]\!]$)

Soient expr une expression de IExpr et $\mathcal{V}$ un élément de IEnvVar.
Si on définit l'ordre $\sqsubseteq^{ef}$ "point à point" sur IEnvFonc par

$$\mathcal{F}_1 \sqsubseteq^{ef} \mathcal{F}_2 \text{ si et seulement si } \forall f \in IFonc\ (\mathcal{F}_1 f) \sqsubseteq^{>} (\mathcal{F}_2 f),$$

alors la fonction

$$\begin{array}{rccc} \rho : & IEnvFonc & \longrightarrow & D \\ & \mathcal{F} & \rightsquigarrow & \varepsilon[\![expr]\!]\mathcal{F}\mathcal{V} \end{array}$$

est continue pour les ordres $\sqsubseteq^{ef}$ et $\sqsubseteq$.

Preuve :

Soit une chaîne $(\mathcal{F}_i)$ de *IEnvFonc*.
La définition de la continuité d'une fonction permet de réécrire la thèse sous la forme :

$$\rho(\bigsqcup_i^{ef} \mathcal{F}_i) = \bigsqcup_i (\rho \mathcal{F}_i).$$

Cette égalité se traduit directement par

$$\varepsilon[\![expr]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V} = \bigsqcup_i (\varepsilon[\![expr]\!]\mathcal{F}_i\mathcal{V}).$$

On prouve cette dernière égalité par induction sur la structure de l'expression $expr$.

- $expr = b$ ou $expr = x$

  Dans ces cas, on a évidemment l'égalité puisque l'évaluation de l'expression est indépendante de l'environnement de fonction.

- $expr = prim\ e_1, ..., e_n$

$$\varepsilon[\![prim\ e_1, ..., e_n]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V}$$

$$= \widetilde{prim}\ in^n(\varepsilon[\![e_1]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V}, ..., \varepsilon[\![e_n]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V})$$

par définition de la fonction ε

$$= \widetilde{prim}\ in^n(\bigsqcup_i(\varepsilon[\![e_1]\!]\mathcal{F}_i\mathcal{V}), ..., \bigsqcup_i(\varepsilon[\![e_n]\!]\mathcal{F}_i\mathcal{V}))$$

par hypothèse de récurrence sur $expr$

$$= \widetilde{prim}\ in^n(\bigsqcup_i^n(\varepsilon[\![e_1]\!]\mathcal{F}_i\mathcal{V}, ..., \varepsilon[\![e_n]\!]\mathcal{F}_i\mathcal{V}))$$

par définition de la borne supérieure produit

$$= \bigsqcup_i(\widetilde{prim}\ in^n(\varepsilon[\![e_1]\!]\mathcal{F}_i\mathcal{V}, ..., \varepsilon[\![e_n]\!]\mathcal{F}_i\mathcal{V}))$$

puisque $\widetilde{prim}$ et in^n sont continues

$$= \bigsqcup_i(\varepsilon[\![prim\ e_1, ..., e_n]\!]\mathcal{F}_i\mathcal{V})$$

de nouveau, par définition de la fonction ε

- $expr = f\ e_1, .., e_n$

  Ce cas est très similaire au précédent.

$$\varepsilon[\![f\ e_1, ..., e_n]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V}$$

$$= ((\bigsqcup_i^{ef} \mathcal{F}_i)f)\ in^n(\varepsilon[\![e_1]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V}, ..., \varepsilon[\![e_n]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V})$$

par définition de la fonction ε

$$= \bigsqcup_i \Big(((\bigsqcup_i^{ef} \mathcal{F}_i)f)\ in^n(\varepsilon[\![e_1]\!]\mathcal{F}_i\mathcal{V}, ..., \varepsilon[\![e_n]\!]\mathcal{F}_i\mathcal{V})\Big)$$

par l'hypothèse de récurrence sur $expr$
et la continuité des fonctions $(\bigsqcup_i^{ef} \mathcal{F}_i)f$ et in^n

$$= \bigsqcup_i \Big((\bigsqcup_i^{>}(\mathcal{F}_i f))\ in^n(\varepsilon[\![e_1]\!]\mathcal{F}_i\mathcal{V}, ..., \varepsilon[\![e_n]\!]\mathcal{F}_i\mathcal{V})\Big)$$

par définition de la borne supérieure liée à l'ordre $\sqsubseteq^{ef}$

$$= \bigsqcup_i \Big(\bigsqcup_i((\mathcal{F}_i f)\ in^n(\varepsilon[\![e_1]\!]\mathcal{F}_i\mathcal{V}, ..., \varepsilon[\![e_n]\!]\mathcal{F}_i\mathcal{V}))\Big)$$

par définition de la borne supérieure liée à l'ordre $\sqsubseteq^{>}$

$$= \bigsqcup_i((\mathcal{F}_i f)\ in^n(\varepsilon[\![e_1]\!]\mathcal{F}_i\mathcal{V}, ..., \varepsilon[\![e_n]\!]\mathcal{F}_i\mathcal{V}))$$

puisque les deux bornes sont redondantes

$$= \bigsqcup_i(\varepsilon[\![f\ e_1, ..., e_n]\!]\mathcal{F}_i\mathcal{V})$$

par définition de la fonction ε

On a donc bien dans tous les cas, l'égalité désirée.$\diamondsuit$

On peut maintenant passer à la démonstration de la continuité de la transformation d'un environnement de fonctions exigée par un programme.

Proposition 2.2 (Continuité de $\tau[\![p]\!]$)

Soit p appartenant à IProg.
$\tau[\![p]\!]$ *est une transformation continue de* $(IEnvFonc, \sqsubseteq^{ef})$.

Preuve :

On procède par induction sur la longueur de p.

Si $p = \textbf{vide}$, $\tau[\![p]\!]$ revient alors à l'identité sur *IEnvFonc* qui est évidemment continue.

On doit maintenant montrer que, si $\tau[\![p]\!]$ est continue, alors $\tau[\![f(x_1, ..., x_n) = expr\ p]\!]$ est également continue. Posons $p' = (f(x_1, ..., x_n) = expr\ p)$.

Soit $(\mathcal{F}_i)$, une chaîne de *IEnvFonc*. La thèse s'exprime par l'équation

$$\bigsqcup_i^{ef}(\tau[\![p']\!]\mathcal{F}_i) = \tau[\![p']\!](\bigsqcup_i^{ef}\mathcal{F}_i).$$

Ce qui se réécrit "point à point"

$$\forall g \in \mathit{IFonc},\ (\bigsqcup_i^{ef}(\tau[\![p']\!]\mathcal{F}_i))g = (\tau[\![p']\!](\bigsqcup_i^{ef}\mathcal{F}_i))g.$$

Considérons d'abord $g \in \mathit{IFonc}$ tel que $g \neq f$.

$$\begin{aligned}
(\tau[\![p']\!](\bigsqcup_i^{ef}\mathcal{F}_i))g &= (\tau[\![p]\!](\bigsqcup_i^{ef}\mathcal{F}_i))g && \text{par définition de la fonction } \tau \\
&= (\bigsqcup_i^{ef}(\tau[\![p]\!]\mathcal{F}_i))g && \text{par hypothèse de récurrence} \\
&= \bigsqcup_i^{>}((\tau[\![p]\!]\mathcal{F}_i)g) && \text{par définition de la borne supérieure sur } \mathit{IEnvFonc} \\
&= \bigsqcup_i^{>}((\tau[\![p']\!]\mathcal{F}_i)g) && \text{par définition de la fonction } \tau \\
&= (\bigsqcup_i^{ef}(\tau[\![p']\!]\mathcal{F}_i))g && \text{par définition de la borne supérieure sur } \mathit{IEnvFonc}
\end{aligned}$$

Envisageons maintenant la situation en f. D'un côté, on a

$$(\tau[\![p']\!](\bigsqcup_i^{ef}\mathcal{F}_i))f = \lambda v^*.\varepsilon[\![expr]\!](\bigsqcup_i^{ef}\mathcal{F}_i)\mathcal{V}_{v^*},$$

et de l'autre

$$(\bigsqcup_i^{ef}(\tau[\![p']\!]\mathcal{F}_i))f = \bigsqcup_i^{>}((\tau[\![p']\!]\mathcal{F}_i)f) = \bigsqcup_i^{>}(\lambda v^*.\varepsilon[\![expr]\!]\mathcal{F}_i\mathcal{V}_{v^*}) = \lambda v^*.\bigsqcup_i(\varepsilon[\![expr]\!]\mathcal{F}_i\mathcal{V}_{v^*}).$$

Il nous faut montrer que ces deux fonctions sont égales. Si on travaille "point par point", il nous faut donc prouver que pour tout élément v^* de D^*, on a l'égalité

$$\varepsilon[\![expr]\!](\bigsqcup_i^{ef} \mathcal{F}_i)\mathcal{V}_{v^\bullet} = \bigsqcup_i(\varepsilon[\![expr]\!]\mathcal{F}_i\mathcal{V}_{v^\bullet}),$$

ce qui découle directement de la continuité de la fonction ε (cfr proposition 2.1). $\Diamond$

2.3.5 Sémantique des exécutions

Il s'agit juste d'associer à chacune des expressions un élément du domaine de valeur. Pour ce faire, on utilise l'environnement de fonctions associé au programme. Les expressions à évaluer ne contenant aucune variable, l'environnement de variables utilisé n'a aucune importance.

Définition 2.12 (Effet d'une exécution : $\epsilon[\![exec]\!]$)

ϵ : *IExec* $\longrightarrow$ D^* *fournit à partir d'une éxécution la liste des valeurs correspondant aux évaluations des expressions dans l'interprétation du programme.*

$$\epsilon[\![\textbf{let } p \textbf{ in } e_1, ..., e_n]\!] = in^n(\varepsilon[\![e_1]\!]\mu[\![p]\!]\mathcal{V}_\perp, ..., \varepsilon[\![e_n]\!]\mu[\![p]\!]\mathcal{V}_\perp)$$

2.4 Sémantique pour le passage par valeur

La section précédente a précisé d'une façon assez "naturelle" (du moins d'un point de vue mathématique) une sémantique du langage pour le passage par nom des arguments lors des appels de fonctions.

On peut légèrement altérer celle-ci afin qu'elle devienne une sémantique pour le passage par valeur des arguments. Il suffit pour cela que la fonction d'évaluation des expressions traduise le fait qu'un appel de fonction nécessite l'évaluation immédiate de tous les arguments.

Pour ce faire, on conserve la fonction d'évaluation précédente qu'on modifie juste dans les cas où le passage par nom et le passage par valeur divergent au niveau des résultats. Ces situations ne peuvent provenir que de l'existence de fonctions non strictes. Il nous suffit donc de "rendre chaque fonction stricte" , ce qu'on obtiendra en composant chaque fonction avec une "identité stricte".

Nous allons donc définir l'identité stricte que nous insèrerons dans la fonction définie précédemment pour l'évaluation des expressions, à savoir $\varepsilon[\![.]\!]$. Toutes les autres fonctions sémantiques sont conservées.

Mais avant cela, reprécisons la notion de fonction stricte dans notre contexte. Pour $f^n \in [D^n \longrightarrow D]$, on dispose déjà d'une définition (cfr définition 1.14), il nous suffit de l'étendre à $f \in [D^* \longrightarrow D]$. Comme les seules fonctions d'intérêt sont de la forme $f = f^n \circ p^n$, on obtient la définition ci-dessous.

Définition 2.13 (Fonction stricte dans $D^>$)

$f \in D^>$ est stricte en son ième argument si et seulement si

- $f = f^n \circ p^n$,
- $f^n \in [D^n \longrightarrow D]$,
- *f^n est stricte en son ième argument.*

Si une fonction f^* est stricte en tous ses arguments, on dira qu'elle est stricte.

Définition 2.14 (Identité stricte : id_s^n)

La fonction $id_s^n : D^n \longrightarrow D^n$ renvoie $\perp^n$ dès qu'un élément du n-uplet constituant son argument vaut $\perp$.

$$id_s^n(d_1, ..., d_n) = \begin{cases} (d_1, ..., d_n) & \text{si } \forall i \; d_i \neq \perp \\ \perp^n & \text{sinon} \end{cases}$$

Définition 2.15 (Fonction d'évaluation : $\varepsilon_v[\![.]\!]$)

$\varepsilon_v : I\!Expr \longrightarrow I\!EnvFonc \longrightarrow I\!EnvVar \longrightarrow D$ renvoie l'évaluation d'une expression, étant donnés un environnement de fonctions et un environnement de variables, en respectant la sémantique du passage par valeur des arguments.

Soient $\mathcal{V} \in I\!EnvVar$ et $\mathcal{F} \in I\!EnvFonc$.

$$\begin{aligned}
&\varepsilon_v[\![b]\!]\mathcal{V}\mathcal{F} = \tilde{b} \\
&\varepsilon_v[\![x]\!]\mathcal{V}\mathcal{F} = \mathcal{V}(x) \\
&\varepsilon_v[\![prim\ expr_1, ..., expr_n]\!]\mathcal{V}\mathcal{F} = \widetilde{prim}(in^n(\varepsilon_v[\![expr_1]\!]\mathcal{V}\mathcal{F}, ..., \varepsilon_v[\![expr_n]\!]\mathcal{V}\mathcal{F})) \\
&\varepsilon_v[\![f\ expr_1, ..., expr_n]\!]\mathcal{V}\mathcal{F} = (\mathcal{F}f)(in^n(id_s^n(\varepsilon_v[\![expr_1]\!]\mathcal{V}\mathcal{F}, ..., \varepsilon_v[\![expr_n]\!]\mathcal{V}\mathcal{F})))
\end{aligned}$$

Il nous faut cependant nous assurer que cette altération de la fonction d'évaluation ne nuit pas à la continuité de la transformation $\tau[\![p]\!]$ (cfr proposition 2.2). Il suffit pour cela de montrer que l'identité stricte est une fonction continue.

Proposition 2.3 (Continuité de id_s^n)

$id_s^n : D^n \longrightarrow D^n$ est une application continue.

Preuve :

Soit (d^i) une chaîne de D^n. On doit prouver l'égalité

$$\bigsqcup_i^n (id_s^n(d^i)) = id_s^n(\bigsqcup_i^n d^i).$$

Comme (d^i) est une chaîne, on a deux cas à considérer.

- Il existe une composante du vecteur toujours à $\perp$, c'est-à-dire

  $$\exists j \in \{1, ..., n\} \mid \forall i \; id_j^i = \perp.$$

  Dans ce cas, on obtient

  $$\begin{array}{l} \forall i \; id_s^n(d^i) = \perp^n \quad \text{ce qui entraîne} \\ \bigsqcup_i^n (id_s^n d^i) = \perp^n \end{array}$$

  D'un autre côté, on a également que la $j$ème composante de $\sqcup_i^n d^i$ vaut $\perp$ et par conséquent

  $$id_s^n(\bigsqcup_i^n d^i) = \perp^n.$$

- Toutes les composantes "dépassent" $\perp$ à un moment donné, c'est-à-dire

  $$\exists k \mid \forall i \geq k \; id_s^n(d^i) = d^i,$$

  et la thèse est immédiate.

$\diamondsuit$

2.5 Un cas particulier

Dans cette section, on présente une instanciation du langage. C'est pour cette instanciation qu'on fournira par la suite une implémentation CaML. Le terme instanciation signifie simplement qu'on précise des symboles de base et les symboles fonctionnels prédéfinis ainsi que les sémantiques de ceux-ci.

On considère une seule constante **zero** et quatres primitives **pred**, **succ**. **egal** et **si** dont les arités sont données par $\mathcal{A}r(\mathbf{pred}) = 1$, $\mathcal{A}r(\mathbf{succ}) = 1$, $\mathcal{A}r(\mathbf{egal}) = 2$ et $\mathcal{A}r(\mathbf{si}) = 3$.

On peut donc réécrire la syntaxe abstraite du langage.

$$\begin{array}{lll} expr & ::= & \mathbf{zero} \mid x \mid \mathbf{pred}\ expr \mid \mathbf{succ}\ expr \\ & & \mid \mathbf{egal}\ (expr, expr) \mid \mathbf{si}\ (expr, expr, expr) \mid f\ expr^* \\ dec & ::= & f\ x^* = expr \\ prog & ::= & \mathbf{vide} \mid dec\ prog \\ exec & ::= & \mathbf{let}\ prog\ \mathbf{in}\ expr^* \end{array}$$

Il s'agit maintenant de donner à sens à ces symboles. Pour cela on se fixe un domaine de valeurs : $D = I\!N^+ = I\!N \cup \{\perp\}$. C'est un domaine primitif qui possède donc une structure de CPO.

On prend naturellement $\widetilde{\mathbf{zero}} = 0$.

On donne maintenant une signification aux primitives, i.e. on associe à **succ**, **pred**, **egal** et **si** des éléments de $[(I\!N^+)^* \longrightarrow I\!N^+]$.

Pour ce faire, on définit les applications suivantes :

$$\begin{array}{lccll} succ : & I\!N^+ & \longrightarrow & I\!N^+ & \\ & \perp & \rightsquigarrow & \perp & \\ & n & \rightsquigarrow & n+1 & \text{si } n \in I\!N \end{array}$$

$$\begin{array}{lccll} pred : & I\!N^+ & \longrightarrow & I\!N^+ & \\ & \perp & \rightsquigarrow & \perp & \\ & 0 & \rightsquigarrow & \perp & \\ & n & \rightsquigarrow & n-1 & \text{si } n \in I\!N_0 \end{array}$$

$$\begin{array}{lccll} egal : & I\!N^+ \times I\!N^+ & \longrightarrow & I\!N^+ & \\ & (\perp, n) & \rightsquigarrow & \perp & \\ & (n, \perp) & \rightsquigarrow & \perp & \\ & (n, n) & \rightsquigarrow & 1 & \text{si } n \in I\!N \\ & (n_1, n_2) & \rightsquigarrow & 0 & \text{si } n_1, n_2 \in I\!N \text{ et } n_1 \neq n_2 \end{array}$$

$$\begin{array}{lccll} si : & I\!N^+ \times I\!N^+ \times I\!N^+ & \longrightarrow & I\!N^+ & \\ & (\perp, n_1, n_2) & \rightsquigarrow & \perp & \\ & (0, n_1, n_2) & \rightsquigarrow & n_2 & \\ & (n_t, n_1, n_2) & \rightsquigarrow & n_1 & \text{si } n_t \in I\!N_0 \end{array}$$

On peut facilement montrer que ces trois fonctions sont continues.

On remarque que *succ*, *pred* et *egal* sont des fonctions strictes tandis que *si* est stricte en sa première variable mais pas en les deux autres.

Pour définir les sémantiques des trois primitives, il suffit de poser

$$\begin{aligned}\widetilde{\mathbf{pred}} &= pred \circ p^1,\\ \widetilde{\mathbf{succ}} &= succ \circ p^1,\\ \widetilde{\mathbf{egal}} &= egal \circ p^2,\\ \widetilde{\mathbf{si}} &= si \circ p^3.\end{aligned}$$

On a maintenant défini totalement un langage applicatif et ce, d'une part, pour une sémantique respectant le passage par nom des arguments et, d'autre part, pour une sémantique respectant le passage par valeur.

La section suivante se consacre à l'explication d'une implémentation de ces deux sémantiques en CaML.

2.6 Implémentation CaML

Pour alléger la lecture, on fournit ici uniquement le texte des fonctions CaML directement relatives aux fonctions sémantiques définies dans les sections précédentes. Le code des fonctions auxiliaires est, quant à lui, disponible en annexe. On en donne cependant les signatures.

CaML étant un langage fonctionnel, la traduction d'une sémantique dénotationelle y est relativement aisée. Les fonctions CaML se construisent de façons analogues aux fonctions sémantiques. Le point délicat réside dans le fait que CaML procède, lui, par "passage par valeur".

2.6.1 Quelques fonctions auxiliaires

Ce paragraphe rassemble quelques signatures de fonctions auxiliaires CaML utilisées dans les fonctions d'intérêt ainsi que des spécifications informelles de celles-ci. Ces fonctions sont pour la plupart des "classiques CaML".

Pour rappel : la notation `a'` désigne un type quelconque.

```
appl : ('a ---> 'b) ---> 'a list ---> 'b list
```

Cette fonction applique une fonction à une liste d'éléments.

```
ponct : ('a ---> 'b) ---> 'a ---> 'b ---> 'a ---> 'b
```

`ponct f a b` est une fonction égale à `f` en tout point sauf en `a` où elle prend la valeur `b` (équivalent CaML de $f[a/b]$).

```
ponctlist : ('a ---> 'b) ---> 'a list ---> 'b list ---> 'a ---> 'b
```

Cette fonction est l'extension de `ponct` à une liste: ainsi, par exemple, `ponctlist f [a;aa] [b;bb]` est l'équivalent CaML de $f[a/b, aa/bb]$.

2.6.2 Types CaML relatifs à la syntaxe abstraite

Les déclarations de type en CaML permettent une traduction quasi directe de la syntaxe abstraite. Les séquences sont implémentées via des listes. Les variables et les symboles fonctionnels sont tous deux représentés par des éléments de type `string`.

Les éléments de *𝔼xpr* se traduisent en CaML par des objets de type `expr`.

```
type expr = ZERO | VAR of string | SUIV of expr | PRED of expr
          | EGAL of expr*expr
          | SI of expr*expr*expr | FONC of string*(expr list);;
```

Les éléments de *ℙrog* se traduisent en CaML par des objets de type `prog` et ceux de *𝔼xec* par des objets de type `exec`.

```
type prog = PROG of ((string*(string list))*expr) list;;

type exec = EXEC of prog*(expr list);;
```

La vérification syntaxique, i.e. la vérification des règles additionnelles est réalisée par la fonction CaML `verifexec` dont le code est fourni en annexe (il est évident que cette fonction se construit à partir de fonctions CaML correspondant aux fonctions du type "*𝒱ars*" définies au début de ce chapitre).

```
verifexec : exec ---> bool
```

Cette fonction renvoie `true` si et seulement si le programme représenté par l'objet CaML de type `prog` est acceptable.

Afin de mieux spécifier les fonctions CaML correspondant aux fonctions sémantiques, on introduit une "méta-fonction" qui associe un objet "du monde mathématique" à un objet "du monde CaML". Cette "méta-fonction" sera symbolisée par $(.)^c$. Ainsi, si x est un élément de $\mathbb{V}ar$, on note sa représentation CaML x^c et on écrit abusivement $(.)^c : \mathbb{V}ar \longrightarrow$ `string`.

On résume ainsi les correspondances introduites jusqu'à présent :

$$\begin{array}{lll} (.)^c : & \mathbb{V}ar & \longrightarrow \texttt{string}, \\ (.)^c : & \mathbb{F}onc & \longrightarrow \texttt{string}, \\ (.)^c : & \mathbb{E}xpr & \longrightarrow \texttt{expr}, \\ (.)^c : & \mathbb{P}rog & \longrightarrow \texttt{prog}, \\ (.)^c : & \mathbb{E}xec & \longrightarrow \texttt{exec}. \end{array}$$

On peut alors spécifier la fonction CaML `verifexec` par

$$\forall exec \in \mathbb{E}xec,\ (\texttt{verifexec}\ exec^c = \texttt{true}) \Leftrightarrow (\vdash\ exec)$$

2.6.3 Fonctions CaML pour le passage par valeur

CaML étant lui-même implémenté selon une sémantique du passage par valeur, il est plus aisé de l'utiliser pour réaliser une implémentation de notre langage suivant le passage par valeur ... C'est pourquoi nous inversons l'ordre de présentation suivi jusqu'ici.

2.6.3.1 Domaine de valeurs et primitives

La première étape de l'implémentation réside dans l'expression du domaine de valeurs et des sémantiques des primitives en "termes CaML".

Le domaine de valeurs $\mathbb{N}^+$ est traduit par le type CaML `val`. Attention, le `BOT` ci-dessous traduit seulement les cas d'indétermination, les cas de non-terminaison étant "représentés" par une non-terminaison effective des fonctions CaML associées aux fonctions sémantiques. On écrit donc

$$(.)^c : \mathbb{N}^+ \longrightarrow \texttt{val+?},$$

où ? représente la non-terminaison.

```
type val = NAT of int | BOT;;
```

Les primitives[3] sont, quant à elles, traduites par les fonctions CaML ci-dessous qui sont des transcriptions directes des fonctions *succ*, *pred*, *egal* et *si* explicitées précédemment.

```
let succ = function
      BOT -> BOT
      | (NAT n) -> (NAT (n+1));;

let pred = function
      BOT -> BOT
      |(NAT 0) -> BOT
      |(NAT n) -> (NAT (n-1));;

let eg = function
      (BOT,_) -> BOT
      | (_,BOT) -> BOT
      | (NAT x, NAT y) -> if x=y then (NAT 1) else (NAT 0);;

let si = function
      (BOT,_,_) -> BOT
      |(NAT 0,x,y) -> y
      |(t,x,y) -> x;;
```

Concrètement, on ne pourra pas utiliser la dernière fonction présentée. En effet, comme CaML applique le passage par valeur, l'appel à cette fonction nécessitera l'évaluation des trois arguments et, dans les cas de non-terminaison d'un des deux derniers arguments, ne fournira pas nécessairement le résultat voulu. On doit, en fait recourir, à la fonction "lazy" `if then else` de CaML (comme on le verra lors de la traduction de la fonction d'évaluation d'une expression).

On peut donc écrire les spécifications :

$$\forall n \in I\!N^+,\ \texttt{succ}\ n^c = (succ(n))^c$$
$$\forall n \in I\!N^+,\ \texttt{pred}\ n^c = (pred(n))^c$$
$$\forall n_1, n_2 \in I\!N^+,\ \texttt{egal}(n_1, n_2)^c = (egal(n_1, n_2))^c$$

2.6.3.2 Environnements

Il nous faut maintenant associer des objets CaML aux environnements de fonctions et aux environnements de variables. Un objet de type `venv` implémente un élément de $I\!Env\!Var$ et un objet de type `fenv` un élément de $I\!EnvFonc$.

[3]Ou plus exactement les fonctions sémantiques associées aux primitives. On appliquera souvent cet abus de dénomination par la suite.

```
type venv = VENV of (string -> val);;

type fenv = FENV of string->((val list) -> val);;
```

Pratiquement, pour pouvoir directement appliquer les fonctions, on laisse tomber les constructeurs.

$$(.)^c : \mathbb{E}nvFonc \longrightarrow \texttt{venv}$$
$$(.)^c : \mathbb{E}nvVar \longrightarrow \texttt{venf}$$

2.6.3.3 Expressions

Passons maintenant à l'implémentation des fonctions sémantiques à proprement parler. La fonction d'évaluation ε_v est implémentée par la fonction `eval`. Pour construire celle-ci, on transpose directement la définition selon la structure des expressions de ε_v via un simple "pattern matching". Pour mieux profiter de la currification, on modifie cependant l'ordre de la signature :

$$\varepsilon_v : \mathbb{E}xpr \longrightarrow \mathbb{E}nvFonc \longrightarrow \mathbb{E}nvVar \longrightarrow D$$

devient

```
eval : fenv ---> venv ---> expr ---> val
```

Ainsi, si `f` est de type `fenv` et `v` de type $\mathbb{E}nvVar$, l'expression `eval f v` a pour valeur une fonction CaML qui associe à tout élément de type `expr` un élément de type `val`.

On peut spécifier `eval` par

$$\forall \mathcal{F} \in \mathbb{E}nvFonc,\ \forall \mathcal{V} \in \mathbb{E}nvVar,\ \forall expr \in \mathbb{E}xpr,\ \texttt{eval}\ \mathcal{F}^c\ \mathcal{V}^c\ expr^c = (\varepsilon_v[\![expr]\!]\mathcal{F}\mathcal{V})^c.$$

Comme on ne peut pas utiliser la fonction `si` définie plus haut, on introduit un `if then else` pour réaliser l'application d'une fonction non stricte, on aboutit alors au code ci-dessous.

```
let rec eval (FENV fe) (VENV e) = function
      ZERO -> (NAT 0)
      |(VAR x)  -> e x
      |(SUIV expr) -> succ(eval (FENV fe) (VENV e) expr)
```

```
|(PRED expr) -> pred(eval (FENV fe) (VENV e) expr)
|(EGAL (ex1,ex2))
  -> eg (eval (FENV fe) (VENV e) ex1 , eval (FENV fe) (VENV e) ex2)
|(SI (ex1, ex2, ex3))
  -> if (eval (FENV fe) (VENV e) ex1) = BOT
     then BOT
     else
       begin
         if (eval (FENV fe) (VENV e) ex1) = (NAT 1)
         then eval (FENV fe) (VENV e) ex2
         else eval (FENV fe) (VENV e) ex3
       end
|(FONC (s, l))
  -> fe(s) (appl (eval (FENV fe) (VENV e)) l);;
```

Remarquons que, en ce qui concerne le traitement des appels de fonctions, le passage par valeur des arguments s'obtient sans artifice puisque CaML évalue directement la liste d'expressions `l`.

2.6.4 Programmes

Il faut maintenant construire les homologues CaML des fonctions τ et μ : `trans` et `modele`. On devrait donc avoir les signatures :

```
trans : prog --->  fenv  ---> fenv
modele : prog ---> fenv
```

En traduisant la fonction $\lambda v^*.\varepsilon[\![expr]\!]\mathcal{FV}_{v^*}$ correspondant à la déclaration $fx_1,...x_n = expr$ par une fonction CaML `fonction expr lv fe` correspondant à la déclaration `((f,lv),expr)`, on peut directement transposer la définition de τ sur `trans`.

On spécifie donc les fonctions CaML `venvnul`, `fonction` et `trans` par :

```
venvnul : envv
fonction : expr ---> string list ---> fenv ---> val list ---> val
```

$$\texttt{venvnul} = (\mathcal{V}_\perp)^c$$

$$\forall expr \in I\!Expr, \forall x_1, ..., x_n \in \mathbb{V}ar, \forall \mathcal{F} \in I\!EnvFonc, \forall n^* \in (I\!N^+)^*,$$
$$\texttt{fonction}\ expr^c\ (x_1, ..., x_n)^c\ \mathcal{F}^c\ (n^*)^c = (\varepsilon_v[\![expr]\!]\mathcal{F}\mathcal{V}_{n^*})^c$$

$$\forall prog \in I\!Prog, \forall \mathcal{F} \in I\!EnvFonc,\ \texttt{trans}\ prog^c \mathcal{F}^c = (\tau[\![prog]\!]\mathcal{F})^c$$

Ce qui se réalise par exemple par le code ci-dessous.

```
let venvnul v = BOT;;

let fonction expr lv fe lval
    = eval (FENV fe) (VENV (ponctlist venvnul lv lval)) expr;;

let rec trans = fun
      (PROG []) fe -> fe
      | (PROG (((f,lv),expr)::p)) fe
      -> ponct (trans (PROG p) fe) f (fonction expr lv fe);;
```

En ce qui concerne la fonction `modele`, on désire obtenir :

$$\forall prog \in I\!Prog,\ \texttt{modele}\ prog^c = (\mu[\![prog]\!])^c$$

Comme l'instruction CaML `let rec` définit en fait le plus petit point fixe d'une transformation, il nous reste simplement à l'appliquer à la transformation `trans` pour construire `modele`.

```
let rec modele p = (trans p (modele p)) ;;
```

2.6.4.1 Exécutions

Le passage de ϵ à son homologue CaML `evaluation` est direct (on lui adjoint juste la fonction de vérification syntaxique).

$$\forall exec \in I\!Exec, \texttt{evaluation}\ exec^c = \begin{array}{l}\text{si } \vdash exec \\ \text{alors } (\epsilon[\![exec]\!])^c \\ \text{sinon afficher un message d'erreur}\end{array}$$

Remarquons qu'ici $I\!Expr$ désigne l'ensemble de base (sans application des règles additionnelles) et non l'ensemble $I\!Expr'$.

```
let evaluation (EXEC (p,e)) =
    if verifexec (EXEC (p,e))
    then appl (eval (FENV (modele p)) (VENV venvnul)) e
    else raise (Erreur "Execution incorrecte");;
```

2.6.5 Fonctions CaML pour le passage par nom

Comment implémenter une sémantique qui respecte le passsage par nom des paramètres? Il faut en fait que lors des évaluations des appels de fonctions, on n'évalue pas directement la liste des arguments mais qu'on évalue chaque argument chaque fois qu'on en a besoin. Comment donc obliger CaML à "différer" l'évaluation de la liste des arguments?

L'idée repose sur le principe que CaML ne calcule réellement que les valeurs simples. Il ne calcule par exemple pas les objets de type fonction (au sens CaML) ce qui est assez logique puisque les fonctions sont des objets infinis et donc non calculables en "extension". On "enveloppe" donc chaque expression correspondant à un argument dans une fonction. Il s'agit d'une fonction constante qu'on applique à un point quelconque lorsqu'on doit effectivement évaluer le paramètre: au lieu d'avoir un argument du style $\varepsilon[\![e_i]\!]\mathcal{FV}$, on a $\lambda x.\varepsilon[\![e_i]\!]\mathcal{FV}$.

On doit évidemment adapter les types CaML relatifs aux environnements à ces considérations puisque les arguments des fonctions ne sont plus des objets de type `val` mais de type `a' ---> val` (on prend arbitrairement `int ---> val`).

```
type venv = VENV of (string -> (int->val));;

type fenv = FENV of string->(((int->val) list) -> val);;
```

De nouveau, pratiquement, on laisse tomber les constructeurs.

Ceci entraîne naturellement une nouvelle définition de la fonction CaML représentant l'environnement de variables nul.

```
let venvnul v x = BOT;;
```

La seule fonction CaML modifiée par rapport à l'interpréteur présenté dans la section précédente pour le passage par valeur est `eval` qui doit traiter les appels de fonctions et les variables selon les considérations évoquées ci-dessus.

La spécification de `eval` devient

$\forall \mathcal{F} \in I\!E\!nvFonc,\ \forall \mathcal{V} \in I\!E\!nvVar,\ \forall expr \in I\!E\!xpr,$ `eval` $\mathcal{F}^c\ \mathcal{V}^c\ expr^c = (\varepsilon[\![expr]\!]\mathcal{FV})^c$.

```
let rec eval (FENV fe) (VENV e) = function
      ZERO -> (NAT 0)
      |(VAR x)  -> (e x) 0
      |(SUIV expr) -> succ(eval (FENV fe) (VENV e) expr)
      |(PRED expr) -> pred(eval (FENV fe) (VENV e) expr)
      |(EGAL (ex1,ex2))
        -> eg (eval (FENV fe) (VENV e) ex1 , eval (FENV fe) (VENV e) ex2)
      |(SI (ex1, ex2, ex3))
        -> if (eval (FENV fe) (VENV e) ex1) = BOT
           then BOT
           else
             begin
               if (eval (FENV fe) (VENV e) ex1) = (NAT 1)
               then eval (FENV fe) (VENV e) ex2
               else eval (FENV fe) (VENV e) ex3
             end
      |(FONC (s, l))
        -> fe(s) (enveloppe (FENV fe) (VENV e) l)
and
      envel fe e expr x = eval fe e expr

and
      enveloppe fe e l = appl (envel fe e) l;;
```

Expliquons plus en détails, les deux nouvelles fonctions `envel` et `enveloppe`.

```
envel : fenv ---> venv ---> expr ---> int ---> val
enveloppe : fenv ---> venv ---> expr list ---> (int ---> val) list
```

La fonction `envel` correspondra simplement à la spécification

$$\forall expr \in I\!E\!xpr,\ \forall \mathcal{F} \in I\!E\!nvFonc,\ \forall \mathcal{V} \in I\!E\!nvVar,$$
$$\texttt{envel}\ \mathcal{F}^c\ \mathcal{V}^c\ expr^c = \texttt{(i : int) ->}\ (\varepsilon[\![expr]\!]\mathcal{F}\mathcal{V})^c$$

La fonction `enveloppe` réalise la même opération mais sur une liste d'expressions.

2.7 Quelques chiffres

Le passage par valeur est, comme nous l'avons déjà dit, beaucoup plus efficace que le passage par nom et ce tant au point de vue temps qu'au point de vue espace. Cette section illustre cette différence par quelques chiffres.

2.7.1 Exemple 1

Considérons le programme[4] suivant.

```
let p1 =

PROG
[(("add",["x";"y"]),
  SI (
      EGAL (VAR "x",ZERO),
      VAR "y",
      SUIV (FONC ("add",[PRED(VAR "x");VAR "y"]))
     )
  );
  (("grand",["a";"b";"c";"d";"e";"f"]),
     FONC ("add",
     [FONC ("add",[
                   FONC ("add",[VAR "a";VAR "b"]);
                   FONC ("add",[VAR "c";VAR "d"])
                   ]
            );
      FONC ("add",[VAR "e";VAR "f"])
     ])
   )
];;
```

Si on retranscrit ce programme sous une forme plus agréable, on obtient :

$$p_1 : \begin{cases} add(x,y) = & \textbf{si } x = 0 \\ & \textbf{alors } y \\ & \textbf{sinon } add(x-1,y)+1 \\ \\ grand(a,b,c,d,e,f) = add(add((add(a,b),add(c,d))),add(e,f)) \end{cases}$$

On constate que les significations attendues des déclarations sont assez évidentes et qu'il s'agit de deux fonctions strictes. Les deux sémantiques sont donc équivalentes. On peut donc effectuer des comparaisons de "temps d'exécution" entre les deux.

[4]Le mot programme est ici à prendre au sens de la représentation CaML d'un élément de *IProg*.

Si considère des exécutions (syntaxiques) de la forme

```
EXEC (p1,[FONC ("grand",[expr;expr;expr;expr;expr;expr])])
```

c'est à dire $grand(expr, expr, expr, expr, expr, expr)$, on obtient le tableau[5] de "temps d'éxécution" (exprimés en seconde) suivant

"$\varepsilon[\![expr]\!]$"	*nom*	*valeur*
1	0	0
2	2	0
3	6	0
4	12	0
5	24	0
6	42	0
7	69	0
8	–	0
50	–	1
100	–	2
200	–	4
1000	–	24
2000	–	51
2400	–	–

Le symbole "–" signifie que le programme n'a pas fourni de résultat (dépassement de la mémoire conventionnelle).

2.7.2 Exemple 2

Considérons les simples opérations arithmétiques + et × ainsi que la traditionnelle factorielle construites comme indiqué dans le programme ci-après.

[5]Pour obtenir ces approximations de temps d'éxécution, on a produit des versions compilées (via CaMLC) des programmes CaML incluant les fonctions de test. On a ensuite inséré les exécutables dans des fichiers BATCH se référant automatiquement à l'horloge interne. Ces tests ont été effectués sur un PC 486/dx33.

$$p_2 : \begin{cases} add(x,y) = & \textbf{si } x = 0 \\ & \textbf{alors } y \\ & \textbf{sinon } add(x-1,y)+1 \\ \\ fois(x,y) = & \textbf{si } x = 0 \\ & \textbf{alors } 0 \\ & \textbf{sinon } add(fois(x-1,y),y) \\ \\ fact(n) = & \textbf{si } x = 0 \\ & \textbf{alors } 1 \\ & \textbf{sinon } fois(n, fact(n-1)) \end{cases}$$

Dans cet exemple, toutes les fonctions ne sont pas strictes. En effet, $fois$ n'est pas stricte en sa première variable puisque $fois(0, \perp) = 0$. Cela dit, si on s'assure de ne pas aboutir à des situations de ce type, on peut quand même comparer les résultats pour les deux interpréteurs.

Le tableau suivant reprend des approximations de temps d'exécution pour des expressions à évaluer du type $fois(expr, expr)$ tandis que le dernier tableau concerne des expressions du type $fact(expr)$ (toujours exprimées en secondes).

"$\varepsilon[\![expr]\!]$"	nom	$valeur$
10	1	0
20	5	1
30	17	2
40	38	3
50	–	5
100	–	21
150	–	50
200	–	–

"$\varepsilon[\![expr]\!]$"	nom	$valeur$
2	0	0
3	4	0
4	–	0
5	–	0
6	–	2
7	–	13
8	–	–

On peut construire de nombreux autres exemples du même type qui confirmeront tous l'efficacité de loin supérieure du passage par valeur relativement au passage par nom.

Chapitre 3

Sémantique abstraite

Le chapitre précédent a précisé exactement le langage applicatif objet de notre analyse. Il a également illustré à quel point le passage par valeur est plus efficace que le passage par nom.

Notre objectif est maintenant de déterminer quand on peut remplacer, en toute sécurité, le passage par nom par le passage par valeur. Comme nous l'avons déjà dit, pour ce faire, nous allons utiliser le principe de "strictness analysis" de Mycroft [9]. Il s'agit de déterminer a priori si les fonctions déclarées sont strictes on non. Si une fonction est stricte, on peut abandonner le passage par nom au profit du passage par valeur.

L'optique de ce chapitre est de replacer cette analyse dans un cadre un peu plus large. Pour étudier statiquement des propriétés de programme, l'idée est souvent de passer à un domaine de valeurs "plus simple". On parle alors de domaine abstrait par opposition au domaine de valeurs "réel" que l'on nomme domaine concret. Il s'agit alors de donner une sémantique aux objets syntaxiques dans ce domaine abstrait. Nous parlerons naturellement de "sémantique abstraite".

Nous utilisons, comme c'est souvent le cas, une sémantique abstraite "homomorphique" à la sémantique "concrète". Par "homorphique", on entend une sémantique construite de manière identique à la sémantique concrète, les deux sémantiques ne différant finalement que par le domaine de valeurs utilisé. Soulignons cependant que si cette démarche est très courante, elle est loin d'être la seule et ne s'adapte pas du tout à certains problèmes (cfr [5]).

L'idée "d'homomorphisme" sous-entend ici également qu'on désire retrouver sur le domaine abstrait la même structure que sur le domaine concret (il s'agit ici de la structure d'ordre partiel complet). De nouveau, soulignons que cette idée usuelle ne doit pas être une règle essentielle.

Ce chapitre se compose de trois parties. La première précise la sémantique abstraite

du langage. La deuxième relie cette dernière à la sémantique abstraite par l'introduction d'une "fonction d'abstraction" et introduit le concept d'abstraction sûre d'une fonction. La troisième partie, la plus importante, dérive de ces concepts des propositions de "correction".

3.1 Présentation de la sémantique

Comme on l'a dit dans l'introduction, cette sémantique est absolument analogue à la sémantique concrète (à savoir la sémantique donnée pour le passage par nom dans le chapitre précédent). Le but de cette section est donc simplement de préciser les notations propres à cette sémantique.

On adopte, de façon naturelle, le même schéma de présentation que dans le chapitre précédent. On laisse cependant tomber les exécutions qui ne nous sont d'aucune utilité puisque notre démarche s'insère dans une analyse statique des programmes.

3.1.1 Sémantique des constantes

Un atome a est interprété comme un élément d'un domaine de valeurs A noté $a^{\#}$.

Le domaine de valeurs A muni de la relation $\leq$ possède une structure d'ordre partiel complet dont l'élément minimum est symbolisé par $\perp\!\!\!\perp$. Les bornes supérieures sont quant à elles notées $\bigvee$.

3.1.2 Sémantique des primitives

Comme pour le domaine concret, on construit systématiquement des ordres partiels complets du type "produit", "somme" et "ensemble des fonctions continues" (cfr propositions 1.4, 1.6 et 1.9).

Espace	*relation*	*minimum*	*borne sup.*
A^n	$\leq^n$	$\perp\!\!\!\perp^n$	$\bigvee^n$
A^*	$\leq^*$	$\perp\!\!\!\perp^*$	$\bigvee^*$
$A^>$	$\leq^>$	$\perp\!\!\!\perp^>$	$\bigvee^>$

Une primitive *prim* est interprétée comme un élément de $A^> = [A^* \longrightarrow A]$ de la forme $prim^{\#} = prim_n^{\#} \circ p_a^n$ (où p_a^n représente l'application de projection $A^* \longrightarrow A$ et où $prim_n^{\#}$ appartient à $[A^n \longrightarrow A]$). On dénommera abusivement ces fonctions "primitives abstraites".

3.1.3 Sémantique des expressions

On définit des "environnements abstraits" pour les variables et les symboles fonctionnels.

Définition 3.1 ($\mathbb{E}nvVar^{\#}$)

$\mathbb{E}nvVar^{\#} = \mathbb{V}ar \longrightarrow A$ *est l'ensemble des environnements abstraits pour les variables.*

De nouveau, on peut reporter la structure de CPO de A sur $\mathbb{E}nvVar^{\#}$ (cfr proposition 1.10). On note l'élément minimum $\mathcal{V}^{\#}_{\perp}$.

Définition 3.2 ($\mathbb{E}nvFonc^{\#}$)

$\mathbb{E}nvFonc^{\#} = \mathbb{F}onc \longrightarrow A^{>}$ *est l'ensemble des environnements abstraits pour les symboles fonctionnels.*

Cet ensemble bénéficie également de la structure de CPO de son image.

On construit l'équivalent abstrait de $\varepsilon[\![.]\!]$ de manière totalement homomorphique.

Définition 3.3 (Fonction d'évaluation abstraite: $\varepsilon^{\#}$)

$\varepsilon^{\#} : \mathbb{E}xpr \longrightarrow \mathbb{E}nvFonc^{\#} \longrightarrow \mathbb{E}nvVar^{\#} \longrightarrow A$ *fournit la valeur abstraite d'une expression étant donnés un environnement de fonctions abstrait et un environnement de variables abstrait.*

Soient $\mathcal{V}^{\#} \in \mathbb{E}nvVar^{\#}$ et $\mathcal{F}^{\#} \in \mathbb{E}nvFonc^{\#}$.

$$\varepsilon^{\#}[\![b]\!]\mathcal{V}^{\#}\mathcal{F}^{\#} = b^{\#}$$
$$\varepsilon^{\#}[\![x]\!]\mathcal{V}^{\#}\mathcal{F}^{\#} = \mathcal{V}^{\#}(x)$$
$$\varepsilon^{\#}[\![prim\ expr_1, ..., expr_n]\!]\mathcal{V}^{\#}\mathcal{F}^{\#} = prim^{\#}(in^n_a(\varepsilon^{\#}[\![expr_1]\!]\mathcal{V}^{\#}\mathcal{F}^{\#}, ..., \varepsilon^{\#}[\![expr_n]\!]\mathcal{V}^{\#}\mathcal{F}^{\#}))$$
$$\varepsilon^{\#}[\![f\ expr_1, ..., expr_n]\!]\mathcal{V}^{\#}\mathcal{F}^{\#} = (\mathcal{F}^{\#}f)(in^n_a(\varepsilon^{\#}[\![expr_1]\!]\mathcal{V}^{\#}\mathcal{F}^{\#}, ..., \varepsilon^{\#}[\![expr_n]\!]\mathcal{V}^{\#}\mathcal{F}^{\#}))$$

3.1.4 Sémantique des programmes

On définit de nouveau "la sémantique abstraite" d'un programme comme le plus petit point fixe d'une transformation de l'espace des environnements de fonctions abstraits.

Définition 3.4 ($\tau^{\#}[\![p]\!]$)

$\tau^{\#}$: $\mathbb{P}rog \longrightarrow \mathbb{E}nvFonc^{\#} \longrightarrow \mathbb{E}nvFonc^{\#}$ fournit la transformation "exigée" par un programme.

Soit $\mathcal{F}^{\#} \in \mathbb{E}nvFonc^{\#}$.

$$\tau^{\#}(\textbf{vide})\mathcal{F}^{\#} = \mathcal{F}^{\#}$$
$$\tau^{\#}(f\ (x_1, ..., x_n) = expr\ prog)\mathcal{F}^{\#} = (\tau[\![prog]\!]\mathcal{F}^{\#})[f/\lambda a^*.\varepsilon^{\#}[\![expr]\!]\mathcal{F}^{\#}\mathcal{V}^{\#}_{a^*}]$$

où $\mathcal{V}^{\#}_{a^*}$ est défini par

$$\mathcal{V}^{\#}_{a^*} = \mathcal{V}^{\#}_{\perp}[x_1/a_1, ..., x_n/a_n] \text{ avec } p^n_a(a^*) = (v_1, ..., v_n).$$

Définition 3.5 (Sémantique abstraite d'un programme : $\mu^{\#}[\![prog]\!]$)

$\mu^{\#}$: $\mathbb{P}rog \longrightarrow \mathbb{E}nvFonc^{\#}$ fournit le plus petit point fixe de la transformation $\tau^{\#}[\![prog]\!]$, dénommé "sémantique abstraite de prog".

Remarquons que la preuve donnée dans le chapitre précédent pour la continuité de $\tau[\![prog]\!]$ est évidemment encore d'application.

3.2 Liens entre sémantique concrète et abstraite

Le lien entre domaine concret et domaine abstrait est réalisé via une fonction $\alpha : D \longrightarrow A$ appelée fonction d'abstraction. On demande que cette fonction soit continue.

On peut donc directement relier les sémantiques abstraite et concrète des atomes :

$$\forall b \in \mathbb{B}ase,\ b^{\#} = \alpha(\tilde{b}).$$

A partir de α, on peut directement construire les fonctions d'abstractions suivantes :

$$\begin{array}{rccc} \alpha^n : & D^n & \longrightarrow & A^n \\ & (d_1, ..., d_n) & \rightsquigarrow & (\alpha(d_1), ..., \alpha(d_n)) \end{array}$$

$$\begin{array}{rccc} \alpha^* : & D^* & \longrightarrow & A^* \\ & \perp^* & \rightsquigarrow & \perp\!\!\!\perp^* \\ & (n, d^n) & \rightsquigarrow & (n, \alpha^n(d^n)) \end{array}$$

Par les propositions 1.5 et 1.8, ces deux fonctions sont également continues.

On dispose donc d'un lien entre les objets de D et de A, de D^n et de A^n et de D^* et de A^*. Pour associer primitives concrètes et primitives abstraites, il faudrait de la même façon pouvoir relier les éléments de $D^>$ et $A^>$. Si on considère f appartenant à $D^>$ et $f^\#$ appartenant à $A^>$, on est en fait dans la situation

$$\begin{array}{ccc} D^* & \xrightarrow{f} & D \\ {\scriptstyle \alpha^*}\downarrow & & \downarrow{\scriptstyle \alpha} \\ A^* & \xrightarrow{f^\#} & A \end{array}$$

Si $f^\#$ est l'abstraction de f, on aimerait que ce diagramme soit commutatif, c'est-à-dire que

$$f^\# \circ \alpha^* = \alpha \circ f.$$

En effet, l'égalité ci-dessus signifie intuitivement que passer aux domaines abstraits avant ou après application de la fonction conserve la même quantité d'information.

Malheureusement, cette propriété, appelée exactitude, est beaucoup trop exigeante : on ne trouvera certainement pas "d'abstraction exacte" pour tous les éléments de $D^>$. Remarquons qu'elle impose que si deux points ont la même abstraction leurs images doivent également avoir la même abstraction.

$$\alpha^*(d_1^*) = \alpha^*(d_2^*) \Rightarrow \alpha(f(d_1^*)) = \alpha(f(d_2^*))$$

On introduit donc un concept plus faible : celui d'abstraction sûre d'une fonction. Mais illustrons, au préalable, ces considérations par un exemple.

3.2.1 Exemple : étude des signes

Ce paragraphe insère les notions mentionnées précédemment dans un exemple classique, celui de l'étude des signes.

On considère comme domaine concret le domaine primitif $\mathbb{Z}^+$. Les propriétés d'intérêt sont les signes des éléments de cet ensemble. On introduit donc naturellement le domaine abstrait et la fonction d'abstraction suivants :

$$A = \{p, n\}^+ = \{\ \perp\!\!\!\perp, p, n\}$$

$$\begin{array}{lccll} \alpha : & \mathbb{Z}^+ & \longrightarrow & \{\ \perp\!\!\!\perp, p, n\} & \\ & \perp & \rightsquigarrow & \perp\!\!\!\perp & \\ & z & \rightsquigarrow & p & \text{si } z \in \mathbb{N}_0 \\ & z & \rightsquigarrow & n & \text{sinon} \end{array}$$

On s'intéresse alors à deux fonctions correspondant aux opérations arithmétiques usuelles de multiplication et d'addition (on vérifie aisément qu'il s'agit bien de fonctions continues).

$$
\begin{array}{lccll}
* : & \mathbb{Z}^+ \times \mathbb{Z}^+ & \longrightarrow & \mathbb{Z}^+ & \\
 & (\bot, z) & \rightsquigarrow & \bot & \\
 & (z, \bot) & \rightsquigarrow & \bot & \\
 & (z_1, z_2) & \rightsquigarrow & z_1 * z_2 & \text{si } z_1, z_2 \in \mathbb{Z}
\end{array}
$$

$$
\begin{array}{lccll}
+ : & \mathbb{Z}^+ \times \mathbb{Z}^+ & \longrightarrow & \mathbb{Z}^+ & \\
 & (\bot, z) & \rightsquigarrow & \bot & \\
 & (z, \bot) & \rightsquigarrow & \bot & \\
 & (z_1, z_2) & \rightsquigarrow & z_1 + z_2 & \text{si } z_1, z_2 \in \mathbb{Z}
\end{array}
$$

On prend comme abstraction de la fonction de multiplication, la fonction $*_a$ définie par le tableau de valeurs ci-dessous.

$*_a$	$\perp\!\!\!\perp$	p	n
$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$
p	$\perp\!\!\!\perp$	p	n
n	$\perp\!\!\!\perp$	n	p

On vérifie facilement que cette fonction, qui traduit simplement la règle usuelle des signes pour la multiplication, est bien une abstraction exacte de la fonction $*$.

Mais considérons maintenant la fonction d'addition. On se rend bien compte que connaître le signe de deux entiers ne suffit pas pour déterminer le signe de leur somme. Pour traduire les cas d'indétermination, on ajoute un élément au domaine abstrait noté $\top\!\!\!\top$ (cet élément signifie "n'importe quel entier ($+$ $\perp\!\!\!\perp$)"). L'ordre sur le domaine abstrait peut se lire : $x \leq y$ signifie "y approxime x", "y englobe x", ... L'introduction d'un nouvel élément nécessite évidemment l'extension de $*_a$.

$*_a$	$\perp\!\!\!\perp$	p	n	$\top\!\!\!\top$
$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$
p	$\perp\!\!\!\perp$	p	n	$\top\!\!\!\top$
n	$\perp\!\!\!\perp$	n	p	$\top\!\!\!\top$

Mais l'introduction de ce nouvel élément ne permet toujours pas de trouver une abstraction exacte de l'opération d'addition. On peut d'ailleurs facilement prouver qu'une telle abstraction n'existe pas. Considérons les points $(-1, 2)$ et $(-3, 2)$ de $\mathbb{Z}^+ \times \mathbb{Z}^+$. Ces

deux points ont la même abstraction (n, p). Si on peut trouver une abstraction exacte, on doit avoir

$$\alpha(+(-1,2)) = \alpha(+(-3,2))$$

$$\alpha(1) = \perp\!\!\!\perp = \top\!\!\!\top = \alpha(-1).$$

Ce qui est évidemment absurde.

On se rend donc compte que l'exactitude est une propriété beaucoup trop contraignante. Cependant, on ne veut pas admettre n'importe quoi comme abstraction. On aimerait au moins garantir que les raisonnements faits sur les valeurs abstraites ne sont pas faux : on ne peut pas dire que la somme d'un nombre négatif et d'un nombre positif est positive. On introduit alors la notion d'abstration sûre qui exprime que le résultat du "raisonnement abstrait" englobe, approxime le résultat exact.

Si on applique cette idée à notre exemple, on obtient que l'abstraction $+_a$ de l'addition doit vérifier la propriété

$$\forall (z_1, z_2) \in \mathbb{Z}^+ \times \mathbb{Z}^+,\ \alpha(+(z_1, z_2)) \leq +_a(\alpha z_1, \alpha z_2).$$

On exprime ainsi quelque part que tout ce que le modèle abstrait dit est vrai (mais on admet qu'il ne dise pas tout).

3.2.2 Abstractions sûres

Le paragraphe précédent a introduit la notion d'abstraction sûre dans le cadre d'un exemple; celui-ci a pour but de définir exactement ce concept.

Définition 3.6 (Abstraction sûre)

Soit f un élément de $D^>$. Une abstraction sûre de f est un element $f^\#$ de $A^>$ qui vérifie

$$\forall d^* \in D^*,\ \alpha(fd^*) \leq f^\#(\alpha^* d^*).$$

Remarquons qu'on peut définir ce concept de manière plus générique.

Définition 3.7 (Abstraction sûre : définition générale)

Soient f une fonction de $D_1 \longrightarrow D_2$ et f_a une fonction de $A_1 \longrightarrow A_2$. On considère les deux fonctions d'abstractions $\alpha_1 : D_1 \longrightarrow A_1$ et $\alpha_2 : D_2 \longrightarrow A_2$. On dira que f_a est une abstraction sûre de f pour α_1 et α_2 si et seulement

$$\forall d_1 \in D_1,\ \alpha_2(fd_1) \leq_2 f_a(\alpha_1 d_1).$$

On demandera que chaque primitive abstraite soit une abstraction sûre de la primitive concrète correspondante. On peut alors se demander si, dans ces conditions, les fonctions sémantiques conserveront cette propriété pour les fonctions définies par un programme; ce qui revient à poser la question de la correction de notre méthode. La section suivante prouve que, moyennant une petite hypothèse supplémentaire, on a bien ce que l'on désire.

Mais avant d'entamer la démonstration de la correction, étendons la notion d'abstraction sûre aux environnements de variables et aux environnements de fonctions.

Définition 3.8 (Abstraction sûre d'un environnement de fonctions)

Soit $\mathcal{F}$ appartenant à $\mathbb{E}nvFonc$. Un élément $\mathcal{F}^\#$ de $\mathbb{E}nvFonc^\#$ est une abstraction sûre de $\mathcal{F}$ si et seulement si pour tout symbole fonctionnel f de $\mathbb{F}onc$, $\mathcal{F}^\# f$ est une abstraction sûre de $\mathcal{F}f$.

Le cas des environnements de variables est un peu différent puisque qu'il n'y est pas à proprement parler question de fonctions. On demande juste que les images par l'environnement abstrait des variables approximent les abstractions des images des variables par l'environnement concret.

Définition 3.9 (Abstraction sûre d'un environnement de variables)

Soit $\mathcal{V}$ appartenant à $\mathbb{E}nvVar$. Un élément $\mathcal{V}^\#$ de $\mathbb{E}nvVar^\#$ est une abstraction sûre de de $\mathcal{V}$ si et seulement si

$$\forall v \in \mathbb{V}ar,\ \alpha(\mathcal{V}v) \leq \mathcal{V}^\# v.$$

3.3 Correction des fonctions sémantiques

Ce qui nous intéresse est de prouver que si les primitives abstraites sont des abstractions sûres des primitives concrètes, on conserve cette propriété pour les fonctions déclarées dans un programme quelconque.

On peut, grâce aux derniers concepts introduits, formuler cela plus précisément.

Supposons que, pour toute constante b de $\mathbb{B}ase$, $b^\#$ soit une abstraction sûre de $\tilde{b}$. Obtient-on alors que, pour tout programme p de $\mathbb{P}rog$, $\mu^\#[\![p]\!]$ est une abstraction sûre de $\mu[\![p]\!]$?

On prouve la correction de la sémantique d'un programme en trois étapes. Chaque étape correspond à une des fonctions sémantiques : $\varepsilon[\![.]\!]$, $\tau[\![.]\!]$ et $\mu_{\_}[\![.]\!]$.

Dans les propositions qui suivent, on omet de mentionner l'hypothèse concernant les primitives exprimée ci-dessus.

3.3.1 Evaluation des expressions

La proposition suivante exprime que si les environnements abstraits sont des abstractions sûres des environnements concrets, l'évaluation abstraite d'une expression approxime l'abstraction de l'évaluation concrète de cette expression.

Proposition 3.1 (Correction de la fonction d'évaluation)

Si $\mathcal{F}$ est un élément de $\mathbb{E}nvFonc$, $\mathcal{V}$ un élément de $\mathbb{E}nvVar$ et e une expression de $\mathbb{E}xpr$ et si $\mathcal{F}^\#$ est une abstraction sûre de $\mathcal{F}$ et $\mathcal{V}^\#$ une abstraction sûre de $\mathcal{V}$, alors les fonctions d'évaluation vérifient l'inégalité

$$\alpha(\varepsilon[\![e]\!]\mathcal{F}\mathcal{V}) \leq \varepsilon^\#[\![e]\!]\mathcal{F}^\#\mathcal{V}^\#.$$

Preuve :

Cette preuve se réalise par induction sur la structure des expressions.

- L'expression est un atome : $e = b$.
  La thèse découle directement du lien entre la sémantique abstraite et la sémantique concrète d'un atome.

$$\alpha(\varepsilon[\![b]\!]\mathcal{F}\mathcal{V}) = \alpha(\tilde{b}) = b^\# = \varepsilon^\#[\![b]\!]\mathcal{F}^\#\mathcal{V}^\#$$

- L'expression est une variable : $e = x$.
  Cette fois, on utilise le fait que $\mathcal{V}^\#$ est une abstraction sûre de $\mathcal{V}$.

$$\alpha(\varepsilon[\![x]\!]\mathcal{F}\mathcal{V}) = \alpha(\mathcal{V}(x)) \leq \mathcal{V}^\#(x) = \varepsilon^\#[\![x]\!]\mathcal{F}^\#\mathcal{V}^\#$$

- L'expression est une application de primitive : $e = prim(e_1, \ldots, e_n)$.
  Développons, pour commencer, l'hypothèse de récurrence.

$$\forall i \in \{1, \ldots, n\},\ \alpha(\varepsilon[\![e_i]\!]\mathcal{F}\mathcal{V}) \leq \varepsilon^\#[\![e_i]\!]\mathcal{F}^\#\mathcal{V}^\#$$

  En appliquant la définition de l'ordre produit sur A^n, on obtient :

$$(\alpha(\varepsilon[\![e_1]\!]\mathcal{F}\mathcal{V}), \ldots, \alpha(\varepsilon[\![e_n]\!]\mathcal{F}\mathcal{V})) \leq^n (\varepsilon^\#[\![e_1]\!]\mathcal{F}^\#\mathcal{V}^\#, \ldots, \varepsilon^\#[\![e_n]\!]\mathcal{F}^\#\mathcal{V}^\#).$$

On utilise maintenant la construction de la fonction α^n.

$$\alpha^n(\varepsilon[\![e_1]\!]\mathcal{FV}, ..., \varepsilon[\![e_n]\!]\mathcal{FV}) \leq^n (\varepsilon^\#[\![e_1]\!]\mathcal{F}^\#\mathcal{V}^\#, ..., \varepsilon^\#[\![e_n]\!]\mathcal{F}^\#\mathcal{V}^\#)$$

On considère, de manière analogue, l'ordre somme sur A^* et la fonction α^*. L'inégalité s'écrit alors :

$$\alpha^*(n, (\varepsilon[\![e_1]\!]\mathcal{FV}, ..., \varepsilon[\![e_n]\!]\mathcal{FV})) \leq^* (n, (\varepsilon^\#[\![e_1]\!]\mathcal{F}^\#\mathcal{V}^\#, ..., \varepsilon^\#[\![e_n]\!]\mathcal{F}^\#\mathcal{V}^\#)).$$

Posons

$$d^* = (n, (\varepsilon[\![e_1]\!]\mathcal{FV}, ..., \varepsilon[\![e_n]\!]\mathcal{FV}))$$

et

$$a^* = (n, (\varepsilon^\#[\![e_1]\!]\mathcal{F}^\#\mathcal{V}^\#, ..., \varepsilon^\#[\![e_n]\!]\mathcal{F}^\#\mathcal{V}^\#)).$$

L 'hypothèse d'induction s'exprime simplement maintenant par

$$\alpha^*(d^*) \leq^* a^*.$$

Revenons à présent au développement général.

$\alpha(\varepsilon[\![prim(e_1, ..., e_n)]\!]\mathcal{FV}) = \alpha(\widetilde{prim}(d^*))$

par définition de la fonction d'évaluation

$\leq prim^\#(\alpha^*(d^*))$

puisque les primitives abstraites sont des abstractions sûres des primitives concrètes

$\leq prim^\#(a^*)$

par l'hypothèse d'induction et la monotonie de $prim^\#$

$= \varepsilon^\#[\![prim(e_1, ..., e_n)]\!]\mathcal{F}^\#\mathcal{V}^\#$

- L'expression est un appel de fonction : $e = f(e_1, ..., e_n)$.
  En reprenant les notations introduites pour le cas précédent, on écrit de nouveau l'hypothèse de récurrence

$$\alpha^*(d^*) \leq^* a^*.$$

$$\begin{aligned}
\alpha(\varepsilon[\![f(e_1,...,e_n)]\!]\mathcal{F}\mathcal{V}) &= \alpha((\mathcal{F}f)d^*) \\
&\leq (\mathcal{F}^\# f)(\alpha^*(d^*)) \\
&\text{car } \mathcal{F}^\# \text{ abstraction sûre de } \mathcal{F} \\
&\leq (\mathcal{F}^\# f)a^* \\
&\text{par l'hypothèse d'induction et la monotonie de } (\mathcal{F}^\# f) \\
&= \varepsilon^\#[\![f(e_1,...,e_n)]\!]\mathcal{F}^\#\mathcal{V}^\#
\end{aligned}$$

La thèse est donc vérifiée pour toutes les expressions de $\mathbb{E}xpr$.$\diamondsuit$

3.3.2 Effet d'un programme sur un environnement

La proposition suivante exprime que si un environnement de fonctions abstrait est une abstraction correcte d'un environnement de fonctions concret, l'application respective des fonctions de transformations abstraite et concrète sur ces environnements conserve l'état de correction.

Proposition 3.2 (Correction de $\tau^\#$)

Si $\mathcal{F}^\#$ est une abstraction sûre de de $\mathcal{F}$ et si $\alpha(\bot) = \bot\!\!\bot$ alors pour tout programme p de $\mathbb{P}rog$, $\tau^\#[\![p]\!]\mathcal{F}^\#$ est une abstraction sûre de $\tau[\![p]\!]\mathcal{F}$.

Preuve :

Il faut donc montrer qui si

$$\forall f \in \mathbb{F}onc,\ \forall d^* \in D^*,\ \alpha((\mathcal{F}f)d^*) \leq (\mathcal{F}^\# f)\ \alpha^*(d^*)$$

alors

$$\forall f \in \mathbb{F}onc,\ \forall d^* \in D^*,\ \alpha(((\tau[\![p]\!]\mathcal{F})f)d^*) \leq ((\tau^\#[\![p]\!]\mathcal{F}^\#)f)\ \alpha^*(d^*).$$

On procède par induction sur la longueur du programme p.

Si le programme est vide ($p =$ **vide**), il suffit d'utiliser l'hypothèse : $\mathcal{F}^\#$ est une abstraction sûre de $\mathcal{F}$.

$$\alpha(((\tau[\![p]\!]\mathcal{F})f)d^*) = \alpha((\mathcal{F}f)d^*) \leq (\mathcal{F}^\# f)\ \alpha^*(d^*) = ((\tau^\#[\![p]\!]\mathcal{F}^\#)f)\ \alpha^*(d^*)$$

Considérons maintenant le programme

$$p = g(x_1, ..., g_n) = e\ p'.$$

Il faut prouver que si la thèse est vérifiée pour p', elle l'est également pour p. On le montre en deux étapes : d'abord, on considère f appartenant à *IFonc* tel que $f \neq g$; ensuite, on envisage le cas de g.

Le cas $f \neq g$ découle directement de l'hypothèse de récurrence.

$$\alpha(((\tau[\![p]\!]\mathcal{F})f)d^*) = \alpha(((\tau[\![p']\!]\mathcal{F})f)d^*)$$

$$\leq ((\tau^\#[\![p']\!]\mathcal{F}^\#)f)\ \alpha^*(d^*) = ((\tau^\#[\![p]\!]\mathcal{F}^\#)f)\ \alpha^*(d^*)$$

Envisageons maintenant la situation en g. D'un côté, on a

$$\alpha(((\tau[\![p]\!]\mathcal{F})g)d^*) = \alpha((\lambda v^*.\varepsilon[\![e]\!]\mathcal{F}\mathcal{V}_{v^*})d^*) = \alpha(\varepsilon[\![e]\!]\mathcal{F}\mathcal{V}_{d^*})$$

et de l'autre

$$((\tau^\#[\![p]\!]\mathcal{F}^\#)f)\ \alpha^*(d^*) = (\lambda a^*.\varepsilon^\#[\![e]\!]\mathcal{F}^\#\mathcal{V}^\#_{a^*})\ \alpha^*(d^*) = \varepsilon^\#[\![e]\!]\mathcal{F}^\#\mathcal{V}^\#_{\alpha^*(d^*)}$$

L'inégalité désirée découlera donc directement de la proposition 3.1 pour peu que $\mathcal{V}^\#_{\alpha^*(d^*)}$ soit une abstraction sûre de $\mathcal{V}_{d^*}$, i.e. si

$$\forall v \in \mathit{Var},\ \alpha(\mathcal{V}_{d^*}(v)) \leq \mathcal{V}^\#_{\alpha^* d^*}(v).$$

- $d^* = \perp^*$
  C'est ici qu'intervient l'hypothèse supplémentaire $\alpha(\perp) = \perp\!\!\!\perp$. En effet, d'un côté on a
  $$\alpha(\mathcal{V}_{\perp^*}(v)) = \alpha(\perp)$$
  et de l'autre
  $$\mathcal{V}^\#_{\alpha^*\perp^*}(v) = \mathcal{V}^\#_{\perp^*}(v) = \perp\!\!\!\perp.$$
  On aura donc bien l'inégalité voulue si on rajoute cette hypothèse.
- $d^* = (j, d^j)$ avec $j \neq n$
  On procède de la même manière que dans le cas précédent.
  $$\alpha(\mathcal{V}_{d^*})(v) = \alpha(\perp) = \perp\!\!\!\perp$$
- $d^* = (n, (d_1, ..., d_n))$
  Si v n'appartient pas à l'ensemble des variables déclarées $\{x_1, ..., x_n\}$, on retombe sur un cas similaire et sinon
  $$\alpha(\mathcal{V}_{d^*}(x_i)) = \alpha(d_i) = \mathcal{V}^\#_{\alpha^* d^*}(x_i)$$
  puisque $\alpha^* d^* = (n, (\alpha d_1, ..., \alpha d_n))$.

On peut donc appliquer la proposition 3.1 et en déduire la thèse. Remarquons qu'en ce qui concerne la dernière inégalité, on prouve en fait l'égalité.$\Diamond$

3.3.3 Sémantique d'un programme

On déduit maintenant la propriété de correction désirée pour la sémantique abstraite d'un programme.

Proposition 3.3 (Correction de $\mu^\#$)

Si $\alpha(\bot) = \bot\!\!\!\bot$, *on a que pour tout programme* p *de* $\mathbb{P}rog$, $\mu^\#[\![p]\!]$ *est une abstraction sûre de* $\mu[\![p]\!]$.

Preuve :

La thèse se développe

$$\forall f \in \mathbb{F}onc,\ \forall d^* \in D^*,\ \alpha((\mu[\![p]\!]f)d^*) \leq (\mu^\#[\![p]\!]f)\ \alpha^*(d^*).$$

La preuve se décompose en quatre étapes principales.

1. **Construction de la suite de Kleene correspondant à** $\mu[\![p]\!]$

   Le premier élément doit être l'élement minimum de $\mathbb{E}nvFonc$.

   $$\mathcal{F}_0 = \lambda f.\bot^>$$

   Les autres éléments sont construits par applications successives de la transformation $\tau[\![p]\!]$.

   $$\mathcal{F}_{i+1} = \tau[\![p]\!]\mathcal{F}_i$$

   La borne supérieure de cette suite n'est autre que $\mu[\![p]\!]$.

   $$\mu[\![p]\!] = \bigsqcup_i^{ef} \mathcal{F}_i$$

   Remarquons que, par définition des ordres "point à point", comme $(\mathcal{F}_i)$ est une chaîne de $\mathbb{E}nvFonc$, pour tout f appartenant à $\mathbb{F}onc$, $(\mathcal{F}f)$ est une chaîne de $D^>$ et pour tout d^* appartenant à D^*, $((\mathcal{F}f)d^*)$ est une chaîne de D.

2. **Construction de la suite de Kleene correspondant à** $\mu^\#[\![p]\!]$

   On procède de manière parfaitement analogue mais dans $\mathbb{E}nvFonc^\#$.

   $$\mathcal{F}_0^\# = \lambda f.\ \bot\!\!\!\bot^>$$

   $$\mathcal{F}_{i+1}^\# = \tau^\#[\![p]\!]\mathcal{F}_i^\#$$

   $$\mu^\#[\![p]\!] = \bigvee_i^{ef} \mathcal{F}_i^\#$$

3. **Etablissement de la propriété pour chaque niveau des suites de Kleene**

   On montre par récurrence que pour tout i, $\mathcal{F}_i^{\#}$ est une abstraction sûre de $\mathcal{F}_i$.

   - $i = 0$
     Soit d^* appartenant à D^*.

     $$\alpha((\mathcal{F}_0 f)d^*) = \alpha(\bot) = \; \perp\!\!\!\perp$$

   - Comme $\mathcal{F}_i^{\#}$ est une abstraction sûre de $\mathcal{F}_i$ (hypothèse de récurrence), on peut appliquer la proposition 3.2. On a donc que $\tau^{\#}[\![p]\!]\mathcal{F}_i^{\#}$ est une abstraction sûre de $\tau[\![p]\!]\mathcal{F}_i$, ce qui est la traduction directe de la thèse : $\mathcal{F}_{i+1}^{\#}$ est une abstraction sûre de $\mathcal{F}_{i+1}$.

4. **Déduction de la thèse**

   Soit d^* un élément de D^*.

   Par le pas précédent de la démonstration, on a

   $$\forall i,\ \alpha((\mathcal{F}_i f)d^*) \leq (\mathcal{F}_i^{\#} f)\ \alpha^* d^*.$$

   ce qui entraîne

   $$\bigvee_i (\alpha((\mathcal{F}_i f)d^*)) \leq \bigvee_i \left((\mathcal{F}_i^{\#} f)\ \alpha^* d^*\right).$$

   Comme $((\mathcal{F}_i f)d^*)$ est une chaîne de l'ordre partiel complet D et que α est monotone $(\alpha((\mathcal{F}_i f)d^*))$ est une chaîne de A et donc, puisque A est un CPO, la première borne supérieure de l'inégalité existe. La seconde a également un sens puisque $((\mathcal{F}_i^{\#} f)\ \alpha^* d^*)$ est aussi une chaîne de A.

   La continuité de la fonction α permet de passer à l'inégalité suivante.

   $$\alpha\left(\bigsqcup_i ((\mathcal{F}_i f)d^*)\right) \leq \bigvee_i \left((\mathcal{F}_i^{\#} f)\ \alpha^* d^*\right)$$

   En utilisant les définitions des bornes supérieures "point à point", on écrit successivement :

   $$\alpha\left((\bigsqcup_i^{>}(\mathcal{F}_i f))d^*\right) \leq \left(\bigvee_i^{>}(\mathcal{F}_i^{\#} f)\right)\ \alpha^* d^*.$$

   $$\alpha\left((\bigsqcup_i^{ef}(\mathcal{F}_i)f)d^*\right) \leq \left(\bigvee_i^{ef}(\mathcal{F}_i^{\#})f\right)\ \alpha^* d^*.$$

La dernière inégalité est l'expression directe de la thèse.$\diamondsuit$

On n'arrive donc bien à ce que l'on désire : si on peut garantir que toutes les primitives abstraites sont des abstractions sûres des primitives concrètes, on peut reporter cette propriété sur les fonctions déclarables dans un programme.

Remarquons qu'on peut démontrer, en suivant presqu'identiquement le schéma de preuve que nous avons utilisé, une propriété analogue pour les abstractions exactes. Cela dit, cette propriété n'est pas d'un grand intérêt puisqu'exiger que toutes les primitives admettent une abstraction exacte est beaucoup trop contraignant.

Chapitre 4

Analyse de "strictness"

Le chapitre précédent a présenté une méthode "générale" d'analyse de propriétés de programmes par l'interprétation abstraite et ce dans le cadre du langage applicatif étudié. Il a également montré qu'on peut facilement obtenir un contexte garantissant la correction de l'analyse.

Ce chapitre applique, quant à lui, les principes présentés précédemment au problème de l'analyse de "strictness". Il s'agit donc dans un premier temps de fournir un domaine abstrait et une fonction d'abstraction correspondant aux propriétés d'intérêt. On retombe alors naturellement sur le concept de fonction stricte introduit dans le premier chapitre.

La construction d'un "analyseur" nécessite de pouvoir trouver des abstractions sûres pour toutes les primitives. Après avoir constaté que certaines abstractions sûres sont préférables à d'autres, on présente un procédé "systématique" d'obtention des "meilleures" abstractions sûres pour les primitives dans le contexte restreint de notre analyse.

On clôture finalement le chapitre par quelques remarques concernant la complétude de la démarche.

4.1 Domaine abstrait et fonction d'abstraction

Rappelons pour commencer le problème qui nous intéresse. On désire, pour des raisons d'efficacité, remplacer le passage par nom des arguments par le passage par le passage par valeur mais seulement, bien entendu, quand ils fournissent les mêmes résultats.

Il s'agit donc de détecter les situations où les deux types de passage de paramètres aboutissent aux mêmes résultats. Comme on l'a déjà dit, il existe une classe de fonctions pour lesquelles on peut garantir cette propriété : les fonctions strictes. Plus précisément, si une fonction est stricte en un argument, on peut alors remplacer le passage par nom

de cet argument par le passage par valeur.

Les propriétés qui nous intéressent sont donc du type :

$$\forall d_2, ..., d_n \in D,\ f(\bot, d_2, ..., d_n) = \bot.$$

L'idée est donc de ne retenir d'un élement du domaine de valeur que l'information : l'élément est-il indéterminé ou non ? On aboutit alors au domaine abstrait et à la fonction d'abstraction :

$$A = \{\ \perp\!\!\!\perp,\ \top\!\!\!\top\},$$

$$\begin{array}{rccl} \alpha : & D & \longrightarrow & A \\ & \bot & \rightsquigarrow & \perp\!\!\!\perp \\ & d & \rightsquigarrow & \top\!\!\!\top \text{ sinon.} \end{array}$$

La relation d'ordre sur A est donnée par $\{(\ \perp\!\!\!\perp,\ \perp\!\!\!\perp), (\ \perp\!\!\!\perp,\ \top\!\!\!\top), (\ \top\!\!\!\top,\ \top\!\!\!\top)\}$.

Les interprétations qui se cachent derrière ces notations sont du style : soit d appartenant à D, si $\alpha(d) = \perp\!\!\!\perp$ on sait que d est "indéterminé" ; si, par contre, $\alpha(d) = \top\!\!\!\top$, cela signifie qu'on ne sait rien sur d. Dans ce sens, on a bien que la propriété couverte par $\top\!\!\!\top$ "englobe" celle couverte par $\perp\!\!\!\perp$.

Il nous faut montrer que la fonction d'abstraction est bien une fonction continue.

Proposition 4.1 (Continuité de α)

La fonction d'abstraction $\alpha : D \longrightarrow A$ définie plus haut est une fonction continue pour les ordres partiels $\sqsubseteq$ et $\leq$.

Preuve :

Soit (d_i) une chaîne de D.

- Tous les éléments de la chaîne valent $\bot$.

$$\begin{array}{l} \forall i\ d_i = \bot \\ \Rightarrow (\forall i\ \alpha d_i = \perp\!\!\!\perp) \wedge (\bigsqcup_i d_i = \bot) \\ \Rightarrow (\alpha(\bigsqcup_i d_i)) = \alpha\bot = \perp\!\!\!\perp = \bigvee_i \perp\!\!\!\perp = \bigvee_i \alpha d_i \end{array}$$

- Il existe un élement de la chaîne à partir duquel, tous les élements sont différents de $\perp$.

$$\begin{aligned}
&\exists j \mid d_j \neq \perp \\
&\Rightarrow (\forall i > j \; d_j \neq \perp) \\
&\Rightarrow (\bigsqcup_i d_i \neq \perp) \wedge (\forall i > j \; \alpha d_i = \top\!\top) \\
&\Rightarrow (\alpha(\bigsqcup_i d_i) = \top\!\top) \wedge (\bigvee_i (\alpha d_i) = \top\!\top)
\end{aligned}$$

Tous les cas étant couverts, on a ce qu'il fallait démontrer.$\Diamond$

4.2 Interprétation du concept d'abstraction sûre

Que signifie dans notre contexte le concept d'abstraction sûre?

Prenons f appartenant à $D^>$ et $f^\#$ appartenant à $A^>$. En toute généralité[1], $f^\#$ est une abstraction sûre de f si et seulement si

$$\forall d^* \in D^*, \; \alpha(fd^*) \leq f^\#(\alpha^* d^*).$$

Dans notre contexte, cela signifie plus précisément

$$\forall d^* \in D^*, (\; f^\#(\alpha^* d^*) = \perp\!\!\!\perp \Rightarrow \alpha(fd^*) = \perp\!\!\!\perp).$$

Ce qu'on peut encore réexprimer par

$$\forall d^* \in D^*, \; (f^\#(\alpha^* d^*) = \perp\!\!\!\perp \Rightarrow fd^* = \perp).$$

En particulier, on a que si une abstraction sûre d'une fonction concrète est stricte (au sens du domaine abstrait), la fonction concrète est également stricte (au sens du domaine concret). Cette expression fait mieux ressortir le sens de "correction" de l'abstraction sûre dans notre analyse. Mais explicitons cette propriété en termes plus exacts.

Proposition 4.2 (Fonctions strictes et abstraction sûre)

Soit f un élément de $D^>$ tel que $f = f_n \circ p^n$ avec f_n appartenant à $[D^n \longrightarrow D]$.
Si la fonction $f^\#$, appartenant à $A^>$, est une abstraction sûre de f et si $f^\#$ est stricte en son $i$ème argument alors f est également stricte en son $i$ème argument.

[1] Du moins pour la définition que nous avons introduite.

Preuve :

On doit en fait simplement montrer que f_n est une fonction stricte en son $i$ème argument. Remarquons que, comme $f^\#$ est stricte, on a $f^\# = f_n^\# \circ p_a^n$ avec $f_n^\#$ appartenant à $A^n \longrightarrow A$. Sans perdre de généralité, on prend $i = 1$.

On procède en deux étapes principales :

- on montre que $f_n^\#$ est une abstraction sûre de f_n pour les fonctions d'abstractions α^n et α,
- on en déduit que f_n est stricte en son premier argument.

Etape 1

Soit d^n appartenant à D^n. L'élément (n, d^n) appartient à D^*. La thèse découle directement du fait que $f^\#$ est une abstraction sûre de f.

En effet, d'un côté on a

$$\alpha(f(n, d^n)) = \alpha(f_n(p^n(n, d^n))) = \alpha(f_n d^n),$$

et de l'autre

$$f^\#(\alpha^*(n, d^n)) = f^\#(n, \alpha^n d^n) = f_n^\#(p_a^n(n, \alpha^n d^n)) = f_n^\#(\alpha^n d^n).$$

Etape 2

Soient $d_2, ..., d_n$ appartenant à D.

$$\begin{aligned}
\alpha(f_n(\perp, d_2, ..., d_n)) &\leq f_n^\#(\alpha^n(\perp, d_2, ..., d_n)) && \text{puisque } f_n^\# \text{ est une abstraction sûre de } f_n \\
&= f_n^\#(\perp\!\!\!\perp, \alpha d_2, ..., \alpha d_n) && \text{par définition de } \alpha^n \\
&= \perp\!\!\!\perp && \text{puisque la fonction } f_n^\# \text{ est stricte en son premier argument}
\end{aligned}$$

On obtient donc que $f_n(\perp, d_2, ..., d_n) = \perp$. Nous pouvons, par conséquent, clore notre démonstration.$\diamondsuit$

A partir de là, on peut réexprimer les théorèmes de correction du chapitre précédent en termes plus adaptés à notre analyse. En particulier, le théorème de correction de la sémantique d'un programme (cfr proposition 3.3) exprime que, si les primitives abstraites sont des abstractions sûres des primitives concrètes et si la fonction d'abstraction vérifie $\alpha(\perp) = \perp\!\!\!\perp$, alors, si la sémantique abstraite d'un symbole fonctionnel est stricte, la sémantique concrète de ce symbole fonctionnel ne peut être que stricte. Ce qui exprime clairement que notre démarche est "correcte" pour peu que l'on s'assure de deux choses :

- $\alpha(\bot) = \perp\!\!\!\perp$ (ce qui est clairement le cas),
- $\forall prim \in I\!Prim$, $prim^{\#}$ est une abstraction sûre de $\widetilde{prim}$.

Il serait donc intéressant de disposer d'un procédé systématique de construction d'abstractions sûres pour les primitives.

4.3 Abstraction sûre de la fonction *si*

Avant de chercher un procédé systématique de construction d'abstractions sûres pour les primitives, considérons le cas particulier et instructif de la primitive *si* qui. par ailleurs, intervient dans notre langage restreint.

Dans cette section, on travaille sur des espaces du type $D^3 \longrightarrow D$, ce qui évite de s'encombrer des opérations de projection et d'injection.

Rappelons la définition de la fonction *si*.

$$\begin{array}{rccll} si: & I\!N^+ \times I\!N^+ \times I\!N^+ & \longrightarrow & I\!N^+ & \\ & (\bot, n_1, n_2) & \rightsquigarrow & \bot & \\ & (0, n_1, n_2) & \rightsquigarrow & n_2 & \\ & (n_t, n_1, n_2) & \rightsquigarrow & n_1 & \text{si } n_t \in I\!N_0 \end{array}$$

Cette fonction fournit un exemple de plus de l'exigence du concept d'abstraction exacte. On peut en effet facilement montrer que, pour notre domaine abstrait, il est impossible de trouver une abstraction exacte de la fonction *si*.

Il nous suffit pour cela de trouver deux point de $(I\!N^+)^3$ ayant la même abstraction et dont les images n'ont, quant à elles, pas la même abstraction :

$$\alpha^3(0, \bot, 1) = (\alpha 0, \alpha\bot, \alpha 1) = (\top\!\!\!\top, \perp\!\!\!\perp, \top\!\!\!\top) = (\alpha 1, \alpha\bot, \alpha 1) = \alpha^3(1, \bot. 1)$$

$$\alpha(si(0, \bot, 1)) = \alpha(1) = \top\!\!\!\top \neq \perp\!\!\!\perp = \alpha(\bot) = \alpha(si(1, \bot, 1))$$

Il est par contre aisé d'en trouver des abstractions sûres : on vérifie très facilement que les fonctions $si_1^{\#}$, $si_2^{\#}$ et $si_3^{\#}$ présentées ci-après conviennent.

$$
\begin{array}{lccl}
si_1^{\#} : & A^3 & \longrightarrow & A \\
 & (a_1, a_2, a_3) & \rightsquigarrow & \top\!\!\top
\end{array}
$$

$$
\begin{array}{lccl}
si_2^{\#} : & A^3 & \longrightarrow & A \\
 & (\perp\!\!\perp, a_2, a_3) & \rightsquigarrow & \perp\!\!\perp \\
 & (\top\!\!\top, a_2, a_3) & \rightsquigarrow & \top\!\!\top
\end{array}
$$

$$
\begin{array}{lccll}
si_3^{\#} : & A^3 & \longrightarrow & A & \\
 & (\perp\!\!\perp, a_2, a_3) & \rightsquigarrow & \perp\!\!\perp & \\
 & (\top\!\!\top, \perp\!\!\perp, \perp\!\!\perp) & \rightsquigarrow & \perp\!\!\perp & \\
 & (\top\!\!\top, a_2, a_3) & \rightsquigarrow & \top\!\!\top & \text{sinon}
\end{array}
$$

On "sent" immédiatement que ces trois fonctions ne sont pas "aussi bonnes" les unes que les autres. La première fonction $si_1^{\#}$ est évidemment sûre mais n'amène aucune information. La troisième conserve plus de caractéristiques de la fonction si que la deuxième. Au niveau de l'ordre sur $[A^3 \longrightarrow A]$, on a $si_3^{\#} \leq^{>} si_2^{\#} \leq^{>} si_1^{\#}$ et on retrouve cette idée d'approximation d'un élément par un autre élément plus grand.

Intuitivement, on aimerait prendre l'abstraction "la plus précise" parmi les abstractions sûres, on aimerait conserver dans le monde abstrait le plus d'information possible sur la fonction initiale.

Dans notre exemple, il est évident que toute fonction plus petite que $si_3^{\#}$ n'est plus une abstraction sûre de si et qu'on devra par conséquent "se contenter" de $si_3^{\#}$.

La section suivante propose et développe un procédé systématique permettant de trouver l'équivalent de $si_3^{\#}$ pour toutes les primitives.

4.4 Construction d'abstractions sûres

Cette section fournit un moyen de trouver la "meilleure abstraction sûre" pour toute fonction de $[D^* \longrightarrow D]$. Remarquons que l'existence de ce moyen garantit l'existence d'une abstraction sûre pour toute fonction de $[D^* \longrightarrow D]$.

Soulignons qu'on se trouve dans le cadre restreint du domaine abstrait et de la fonction d'abstraction particuliers définis pour notre analyse.

Pour construire notre "meilleure abstraction sûre" en un point a^* de A^*, on considère tous les points du domaine concret "approximés" par a^* et on prend la borne supérieure des abstractions des images par la fonction concrète de ces points.

Définition 4.1 (Meilleure abstraction sûre)

Soit f appartenant à $D^>$. On définit $f^\#$ la meilleure abstraction sûre de f par

$$f^\# a^* = \bigvee \{\alpha(f d^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a^*\}.$$

Remarquons que cette définition n'a un sens que si on peut, pour tout a^* appartenant à A^*, garantir l'existence de la borne supérieure. Pour ce domaine abstrait simple, qui bénéficie en fait d'une structure de treillis complet, l'existence de la borne supérieure est effectivement garantie.

Il nous faut maintenant montrer que la fonction définie par la définition 4.1 correspond bien à ce que l'on désire. C'est ce que réalisent les propositions suivantes.

Proposition 4.3 (Continuité de la meilleure abstraction sûre)

Soit f appartenant à $D^>$. La fonction $f^\# : A^* \longrightarrow A$ définie par

$$f^\# a^* = \bigvee \{\alpha(f d^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a^*\}$$

est continue.

Preuve :

Comme A^* est fini, on peut appliquer la proposition 1.1. Par conséquent, il suffit de montrer que $f^\#$ est monotone.

Soient a_1^* et a_2^* appartenant à A^* et tels que $a_1^* \leq^* a_2^*$.

Soit d^* appartenant à D^*. Si $\alpha^* d^* \leq^* a_1^*$, par transitivité on a également $\alpha^* d^* \leq^* a_2^*$. Par conséquent,on a l'inclusion suivante

$$\{\alpha(f d^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a_1^*\} \subseteq \{\alpha(f d^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a_2^*\}.$$

Ce qui implique au niveau des bornes supérieurs l'inégalité :

$$f^\# a_1^* = \bigvee \{\alpha(f d^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a_1^*\} \leq \bigvee \{\alpha(f d^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a_2^*\} = f^\# a_2^*.$$

$\Diamond$

Proposition 4.4 (Abstraction sûre)

Soit f appartenant à $D^>$. La fonction $f^\# : A^* \longrightarrow A$ définie par

$$f^\# a^* = \bigvee \{\alpha(fd^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a^*\}$$

est bien une abstraction sûre de f.

Preuve :

Cette proposition est quasiment triviale.

Soit d' un élément de D^*.

$$f^\#(\alpha^* d') = \bigvee \{\alpha(fd^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* \alpha^* d'\}$$

En particulier, on a évidemment $\alpha^* d' \leq^* \alpha^* d'$ et donc $\alpha(fd') \leq f^\#(\alpha^* d')$.$\Diamond$

Proposition 4.5 (Plus petite abstraction sûre)

Soient f appartenant à $D^>$ et f_a appartenant à $A^>$. Si $f_a \leq^> f^\#$, où la fonction $f^\#$ est la fonction définie par la définition 4.1 et si f_a est une abstraction sûre de f, les fonctions f_a et $f^\#$ sont identiques.

Preuve :

Soient a^* un élément de A^* et d^* un élément de D^* tel que $\alpha^* d^* \leq^* a^*$.

Comme f_a est une abstraction sûre de f, on a l'inégalité

$$\alpha(fd^*) \leq f_a(\alpha^* d^*).$$

D'autre part f_a monotone et $f_a \leq^> f^\#$ et donc

$$f_a(\alpha^* d^*) \leq f_a a^* \leq f^\# a^*.$$

On a donc

$$\forall d^* \mid \alpha^* d^* \leq^* a^*,\ \alpha(fd^*) \leq f_a a^* \leq f^\# a^*.$$

Et par conséquent, on peut écrire

$$f^\# a^* = \bigvee \{\alpha(fd^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* \alpha^* d'\} \leq f_a a^* \leq f^\# a^*.$$

$\Diamond$

4.5 Cas particulier

Appliquons les notions présentées dans ce chapitre au cas du langage particulier présenté à la section 2.5. Il s'agit donc simplement de donner une sémantique abstraite pour l'atome **zero** et de calculer les meilleures abstractions sûres des primitives **pred**, **succ**, **egal** et **si**.

On prend naturellement $\mathbf{zero}^{\#} = \top\!\!\top$.

Calculons explicitement la meilleure abstraction de la fonction *egal*.

$$\begin{array}{rccll} egal: & I\!N^{+} \times I\!N^{+} & \longrightarrow & I\!N^{+} & \\ & (\bot, n) & \rightsquigarrow & \bot & \\ & (n, \bot) & \rightsquigarrow & \bot & \\ & (n, n) & \rightsquigarrow & 1 & \text{si } n \in I\!N \\ & (n_1, n_2) & \rightsquigarrow & 0 & \text{si } n_1, n_2 \in I\!N \text{ et } n_1 \neq n_2 \end{array}$$

La fonction abstraite est définie par :

$$egal^{\#}(a_1, a_2) = \bigvee \{\alpha(egal(n_1, n_2)) \mid (n_1, n_2 \in I\!N^{+}) \wedge (\alpha n_1 \leq a_1) \wedge (\alpha n_2 \leq a_2)\}.$$

Point par point, on obtient :

$$egal^{\#}(\perp\!\!\!\perp, \perp\!\!\!\perp) = \alpha(egal(\bot, \bot)) = \alpha(\bot) = \perp\!\!\!\perp$$

$$egal^{\#}(\perp\!\!\!\perp, \top\!\!\top) = \bigvee \{\alpha(egal(\bot, n_2)) \mid (n_2 \in I\!N^{+})\} = \alpha(\bot) = \perp\!\!\!\perp$$

$$egal^{\#}(\top\!\!\top, \perp\!\!\!\perp) = \bigvee \{\alpha(egal(n_1, \bot)) \mid (n_1 \in I\!N^{+})\} = \perp\!\!\!\perp$$

$$egal^{\#}(\top\!\!\top, \top\!\!\top) = \bigvee \{\alpha(egal(n_1, n_2)) \mid (n_1, n_2 \in I\!N^{+})\} = \top\!\!\top$$

On peut calculer de la même manière les meilleures abstractions de *pred*, *succ* et *si* : $pred^{\#}$ et $succ^{\#}$ se résument à l'identité sur A et $si^{\#}$ a déjà été fourni dans la section 4.3.

On dispose maintenant de tous les outils nécessaires à l'élaboration d'un "analyseur sûr" permettant la détection de fonctions strictes dans les programmes écrits dans notre langage applicatif particulier.

Il nous suffira "tout simplement" de calculer la sémantique abstraite dans notre domaine à deux valeurs $\{\perp\!\!\!\perp, \top\!\!\top\}$ des programmes considérés.

Le domaine abstrait étant fini (et petit), on pourra calculer la table complète du plus petit point fixe correspondant à cette sémantique. Mais avant d'expliquer, de manière

"pragmatique", les algorithmes à la base des analyseurs implémentés (ce qui sera l'objet du chapitre suivant), il nous reste un dernier point théorique à aborder: celui de la complétude de notre analyse.

4.6 Remarque sur la complétude

Dans le chapitre précédent, on a abordé la question de la correction: "si le modèle abstrait dit qu'une fonction est stricte, est-elle effectivement stricte?". Une autre question importante est la complétude de la démarche: "toute fonction stricte est-elle détectée comme telle par l'analyse?".

On montre, dans cette section, qu'on peut obtenir un résultat de complétude au niveau des primitives mais qu'il est impossible de transposer ce résultat au niveau des fonctions déclarées dans un programme. En effet, lorsqu'on calcule l'abstraction d'une primitive, on dispose de toute l'information concernant cette primitive. Ce n'est évidemment plus le cas quand on est amené à déterminer la sémantique abstraite d'une fonction déclarée.

4.6.1 Primitives

On peut dire que si une fonction est stricte alors sa meilleure abstraction sûre (cfr définition 4.1) est nécessairement stricte.

Proposition 4.6 (Complétude de la meilleure abstraction)

Soient f une fonction de $D^{>}$ et $f^{\#}$ la fonction de $A^{>}$ définie en 4.1. Si f est stricte en sa ième variable alors $f^{\#}$ est également stricte en sa ième variable.

Preuve:

Il nous faut montrer deux choses:

- la fonction $f^{\#}$ est de la forme $f_n^{\#} \circ p_a^n$ avec $f_n^{\#}$ appartenant à $[A^n \longrightarrow A]$,
- la fonction $f_n^{\#}$ est stricte en sa $i$ème variable.

Etape 1

On montre que la fonction $f_n^{\#}$ définie par

$$\forall a^n \in A^n,\ f_n^{\#} a^n = \bigvee \{\alpha(f_n d^n) \mid (d^n \in D^n) \wedge \alpha^n d^n \leq^n a^n\}$$

convient (c'est-à-dire qu'elle vérifie $f^{\#} = f_n^{\#} \circ p_a^n$ et qu'elle est continue). En ce qui concerne la continuité, la démonstration est absolument similaire à la démonstration de la proposition 4.3. On montre juste que la fonction proposée est de la bonne "forme".

Remarquons que si f est stricte alors f est de la forme $f_n \circ p^n$.

- Si $a^* = \perp\!\!\!\perp^*$, d'un côté on a

$$f^{\#} \perp\!\!\!\perp^* = \alpha(f \perp^*) = \alpha(f_n(p^n \perp^*)) = \alpha(f_n \perp^n)$$

  et d'un autre

$$f_n^{\#}(p_a^n \perp\!\!\!\perp^*) = f_n^{\#}(\perp\!\!\!\perp^n) = \alpha(f_n \perp^n).$$

- Si $a^* = (j, a^j)$ avec $j \neq n$, on retombe sur un cas tout à fait similaire au cas de $\perp\!\!\!\perp^*$.
- Si $a^* = (n, a^n)$, on peut écrire successivement

$$f^{\#} a^* = \bigvee \{\alpha(f d^*) \mid d^* \in D^* \wedge \alpha^* d^* \leq^* a^*\}$$

$$= \bigvee \{\alpha(f_n(p^n(d^*)) \mid (d^* = \perp^*) \vee (d^* = (n, d^n) \wedge \alpha^n d^n \leq^n a^n)\}$$

$$= \bigvee \{\alpha(f_n(p^n(d^*)) \mid d^* = (n, d^n) \wedge \alpha^n d^n \leq^n a^n\}$$

$$= \bigvee \{\alpha(f_n d^n) \mid \alpha^n d^n \leq^n a^n)\}$$

$$= f_n^{\#} a^n = f_n^{\#}(p_a^n a^*)$$

Etape 2

Sans perdre de généralité, on considère la première variable.

Soient $a_2, ..., a_n$ appartenant à A.

$$f_n^{\#}(\perp\!\!\!\perp, a_2, ..., a_n) = \bigvee \{\alpha(f_n(d_1, ..., d_n)) \mid \alpha^n(d_1, ..., d_n) \leq^n (\perp\!\!\!\perp, a_2, ..., a_n)\}$$

Donc, par définition de α^n et de l'ordre produit, on obtient

$$f_n^{\#}(\perp\!\!\!\perp, a_2, ..., a_n) = \bigvee \{\alpha(f_n(\perp, d_2, ..., d_n)) \mid \alpha d_2 \leq a_2, ..., \alpha d_n \leq a_n\}.$$

Comme f_n est stricte en sa première variable, on a directement

$$f_n^{\#}(\ \perp\!\!\!\perp, a_2, ..., a_n) = \alpha(\perp) = \ \perp\!\!\!\perp.$$

On a donc bien tout ce qu'il nous fallait.$\diamondsuit$

On a donc finalement qu'une fonction de la forme $f_n \circ p^n$ est stricte en un argument si et seulement si sa meilleure abstraction sûre est stricte en cet argument.

4.6.2 Fonctions déclarées

En ce qui concerne la correction, on a pu reporter la propriété d'abstractions sûres des primitives sur les fonctions déclarables dans un programme (cfr proposition 3.3). On aimerait naturellement reporter la propriété de complétude de la même manière. Malheureusement, la démarche n'est pas complète. Il suffit pour s'en convaincre de considérer l'exemple ci-dessous.

$$p : \begin{cases} f(x) = 0 \\ \\ g(x) = \textbf{si } f(x) = 0 \textbf{ alors } x \textbf{ sinon } 0 \end{cases}$$

Intuitivement la fonction g est équivalente à l'identité qui est évidemment une fonction stricte. Cependant, notre interprétation ne la détectera pas comme telle puisque la seule information retenue sur f est sa "non-indétermination". Le modèle abstrait ne connaît évidemment pas toutes les valeurs de fx. L'évaluation de $g^{\#}(\ \perp\!\!\!\perp)$ engendrera un calcul du type:

$$\begin{aligned} g^{\#}\ \perp\!\!\!\perp &= si^{\#}(egal^{\#}(f^{\#}\ \perp\!\!\!\perp,\ \top\!\!\!\top),\ \perp\!\!\!\perp,\ \top\!\!\!\top) \\ &= si^{\#}(egal^{\#}(\ \top\!\!\!\top,\ \top\!\!\!\top),\ \perp\!\!\!\perp.\ \top\!\!\!\top) \\ &= si^{\#}(\ \top\!\!\!\top,\ \perp\!\!\!\perp,\ \top\!\!\!\top) = \ \top\!\!\!\top. \end{aligned}$$

En ce sens, il est normal que notre analyse ne soit pas complète puique le modèle abstrait efface certaines distinctions au niveau des valeurs concrètes.

En introduisant une primitive de négation définie par

$$\begin{array}{rccc} non: & I\!N^+ & \longrightarrow & I\!N^+ \\ & \perp & \rightsquigarrow & \perp \\ & 0 & \rightsquigarrow & 1 \\ & n & \rightsquigarrow & 0. \end{array}$$

on peut trouver un exemple à une seule déclaration :

$$p : \left\{ \begin{array}{lll} h(x,y) = & \textbf{si } x = 0 & \\ & \textbf{alors } y & \\ & \textbf{sinon} & \textbf{si } non(x = 0) \\ & & \textbf{alors } y \\ & & \textbf{sinon } 1 \end{array} \right.$$

Intuitivement, la sémantique attendue pour cette fonction est

$$\lambda(x,y).(\text{si } x = \bot \text{ alors} \bot \text{ sinon } y).$$

Il s'agit d'une fonction stricte mais l'analyseur la dira stricte seulement en la première variable. Le modèle n'a pas les moyens de percevoir que tous les cas sont couverts par $x = 0$ et $non(x = 0)$ et on aura un calcul du type

$$\begin{array}{rl} f^\#(\ \top,\ \bot\!\!\bot) & = si^\#(egal^\#(\ \top,\ \top),\ \bot\!\!\bot, si^\#(non^\#(egal^\#(\ \top,\ \top)),\ \bot\!\!\bot,\ \top)) \\ & = si^\#(\ \top,\ \bot\!\!\bot, si^\#(non^\#(\ \top),\ \bot\!\!\bot,\ \top)) \\ & = si^\#(\ \top,\ \bot\!\!\bot, si^\#(\ \top,\ \bot\!\!\bot,\ \top)) = si^\#(\ \top,\ \bot\!\!\bot,\ \top) = \ \top. \end{array}$$

Remarquons qu'on pourrait trouver des domaines abstraits à peine plus complexes (c'est-à-dire un peu plus "riches") qui s'adaptent à nos hypothèses et permettent de détecter que ces fonctions sont strictes. Cela dit, il sera toujours possible de trouver d'autres exemples pour lesquels une partie de l'information nécessaire est perdue par le modèle abstrait ...

Chapitre 5

Construction de "l'analyseur"

Le but de ce dernier chapitre est la construction effective d'un analyseur permettant la détection de fonctions strictes et ce dans le cadre restreint du langage particulier présenté à la section 2.5.

Comme nous l'avons déjà signalé, le principe de base est de profiter du fait que le domaine abstrait est fini pour calculer "en extension" la sémantique abstraite des programmes. On constate cependant que l'application naïve de ce principe engendre des problèmes d'efficacité. On développe donc un deuxième algorithme qui ne tient compte que de certaines valeurs "d'intérêt" de la sémantique abstraite.

Ce chapitre se compose de trois parties principales.

La première explique, de manière informelle et via des exemples, les principes à la base des deux algorithmes, respectivement dénommés "développement complet" et "développement restreint".

La deuxième partie s'attarde sur la description de de l'implémentation CaML de ceux-ci. Elle explique également comment, une fois l'analyseur construit, utiliser les informations obtenues par celui-ci pour modifier l'implémentation de la sémantique du passage par nom présentée au chapitre 2.

La troisième partie donne simplement, par quelques tests, une idée des performances des deux analyseurs et de l'interpréteur "hybride" obtenus au terme de la deuxième partie.

En ce qui concerne les descriptions des implémentations, tout comme pour les deux sémantiques, on ne fournit que le code des fonctions principales; la présentation des fonctions auxiliaires se résumant quant à elle à des spécifications informelles (on trouvera cependant le code de celles-ci en annexe).

5.1 Principes algorithmiques

Comme signalé dans l'introduction, cette section présente les algorithmes utilisés de manière informelle via des exemples. On ne trouvera donc pas, par exemple, de preuve rigoureuse de terminaison.

5.1.1 Développement complet

La première idée est d'appliquer directement le principe de base mentionné précédemment et de calculer la "table" complète du plus petit point fixe correspondant à la sémantique abstraite d'un programme p de $\mathbb{P}rog$.

Pour ce faire, on calcule explicitement chaque élément de la suite de Kleene correspondant à ce point fixe ($\mu^{\#}[\![p]\!]$) :

$$\forall a^* \in A^*,\ (\mathcal{F}_0^{\#} f)a^* = \bot\!\!\!\bot$$

$$\forall a^* \in A^*,\ (\mathcal{F}_{i+1}^{\#} f)a^* = ((\tau^{\#}[\![p]\!]\mathcal{F}_i^{\#})f)a^*$$

L'algorithme s'arrête dès que $\mathcal{F}_{i+1}^{\#} = \mathcal{F}_i^{\#}$.

On atteindra effectivement cette situation à un moment donné puisque le nombre d'éléments de A^n est fini ainsi que le nombre de fonctions déclarées et que la suite de Kleene est croissante (i.e. il s'agit d'une chaîne)[1].

Mais explicitons cette méthode sur un exemple. Considérons le programme suivant

$$p_1 : \begin{cases} f(x,y) = \textbf{si } x = 0 \\ \qquad\qquad\ \textbf{alors } g(x) \\ \qquad\qquad\ \textbf{sinon } f(x-1, f(x,y)) \\ \\ g(x) = = \textbf{si } x = 0 \\ \qquad\qquad\ \textbf{alors } 1 \\ \qquad\qquad\ \textbf{sinon } g(x-1) \end{cases}$$

Les sémantiques attendues pour les deux fonctions déclarées f et g sont respectivement $\lambda(x,y).(\text{si } x = \bot \text{ alors } \bot \text{ sinon } 1)$ et $\lambda x.(\text{si } x = \bot \text{ alors } \bot \text{ sinon } 1)$. On aimerait donc que l'analyseur nous indique que g est stricte et que f est stricte en sa première variable mais pas en sa seconde variable.

[1] Si n est le nombre maximum de variables pour une fonction et p le nombre de déclarations, une borne, très large, du nombre d'itérations est donnée par 2^{np}.

Les itérations du point fixe peuvent se représenter sous la forme des tableaux suivants. Par facilité, on passe directement aux fonctions sur les espaces produits : les valeurs des fonctions en les autres points des espaces sommes étant sans intérêt.

Pour l'initialisation, les fonctions prennent la valeur $\perp\!\!\!\perp$ en tous les points.

$\mathcal{F}_0^\# :$

$f :$

$(\perp\!\!\!\perp, \perp\!\!\!\perp)$	$(\perp\!\!\!\perp, \top\!\!\!\top)$	$(\top\!\!\!\top, \perp\!\!\!\perp)$	$(\top\!\!\!\top, \top\!\!\!\top)$
$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$

$g :$

$\perp\!\!\!\perp$	$\top\!\!\!\top$
$\perp\!\!\!\perp$	$\perp\!\!\!\perp$

Pour passer à l'itération suivante, on évalue les expressions correspondant aux fonctions déclarées dans l'environnement $\mathcal{F}_0$ et ce en tous les points.

$\mathcal{F}_1^\# :$

$f :$

$(\perp\!\!\!\perp, \perp\!\!\!\perp)$	$(\perp\!\!\!\perp, \top\!\!\!\top)$	$(\top\!\!\!\top, \perp\!\!\!\perp)$	$(\top\!\!\!\top, \top\!\!\!\top)$
$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$\top\!\!\!\top$	$\top\!\!\!\top$

$g :$

$\perp\!\!\!\perp$	$\top\!\!\!\top$
$\perp\!\!\!\perp$	$\top\!\!\!\top$

On doit donc appeler 6 fois la fonction d'évaluation. Si on réitère le procédé encore une fois, on constate que $\mathcal{F}_2^\# = \mathcal{F}_1^\#$ et l'algorithme s'arrête. Il faut donc au total 12 évaluations. Remarquons qu'on a bien les résultats attendus.

On peut réaliser une première petite amélioration en se rappelant que la suite de Kleene est une chaîne de $\mathbb{E}nvFonc^\#$ et que par conséquent

$$\forall i,\ (\mathcal{F}_i^\# f) \leq^> (\mathcal{F}_{i+1}^\# f).$$

Ce qui signifie que si à un moment on obtient la valeur $\top\!\!\!\top$, cette valeur ne changera plus par la suite. On passe alors, dans notre exemple, à 9 évaluations.

Le problème majeur de cet algorithme réside dans le fait que le nombre d'évaluations nécessaires explose avec l'augmentation du nombre de variables et du nombre de fonctions (croissance exponentielle). Il suffit pour s'en convaincre de considérer le programme suivant, à une seule déclaration

$$p_2 : \begin{cases} f(a,b,c,d) = & \textbf{si } c = 0 \\ & \textbf{alors } d \\ & \textbf{sinon } f(f(a,b,c,d), f(a,b,c,d), c-1, d) + 1 \end{cases}$$

Les itérations du point fixe à ce programme sont reprises dans le tableau suivant.

a^4	$(\mathcal{F}_0^{\#} f)a^4$	$(\mathcal{F}_1^{\#} f)a^4$	$(\mathcal{F}_2^{\#} f)a^4$
($\perp$, $\perp$, $\perp$, $\perp$)	$\perp$	$\perp$	$\perp$
($\top$, $\perp$, $\perp$, $\perp$)	$\perp$	$\perp$	$\perp$
($\perp$, $\top$, $\perp$, $\perp$)	$\perp$	$\perp$	$\perp$
($\perp$, $\perp$, $\top$, $\perp$)	$\perp$	$\perp$	$\perp$
($\perp$, $\perp$, $\perp$, $\top$)	$\perp$	$\perp$	$\perp$
($\top$, $\top$, $\perp$, $\perp$)	$\perp$	$\perp$	$\perp$
($\top$, $\perp$, $\top$, $\perp$)	$\perp$	$\perp$	$\perp$
($\top$, $\perp$, $\perp$, $\top$)	$\perp$	$\perp$	$\perp$
($\perp$, $\top$, $\top$, $\perp$)	$\perp$	$\perp$	$\perp$
($\perp$, $\top$, $\perp$, $\top$)	$\perp$	$\perp$	$\perp$
($\perp$, $\perp$, $\top$, $\top$)	$\perp$	$\top$	$\top$
($\top$, $\top$, $\top$, $\perp$)	$\perp$	$\perp$	$\perp$
($\top$, $\top$, $\perp$, $\top$)	$\perp$	$\perp$	$\perp$
($\top$, $\perp$, $\top$, $\top$)	$\perp$	$\top$	$\top$
($\perp$, $\top$, $\top$, $\top$)	$\perp$	$\top$	$\top$
($\top$, $\top$, $\top$, $\top$)	$\perp$	$\top$	$\top$

On totalise donc $16 + 12 = 28$ évaluations. L'idée à la base de l'algorithme en développement restreint est que parmi toutes les valeurs de ce tableau seules quelques unes nous sont véritablement utiles.

5.1.2 Développement restreint

On tente de réaliser moins d'évaluations. Pour ce faire on constate que notre analyse n'a effectivement besoin que des valeurs des fonctions abstraites en certains points.

Rappelons qu'une fonction continue $f^{\#} : A^n \longrightarrow A$ est stricte, par exemple, en son premier argument si et seulement si

$$\forall a_2, ..., a_n \in A,\ f^{\#}(\ \perp\!\!\!\perp, a_2, ..., a_n) = \ \perp\!\!\!\perp.$$

Considérons maintenant un point de A^n de la forme $(\ \perp\!\!\!\perp, a_2, ..., a_n)$. Il verifie nécessairement l'inégalité ci-dessous.

$$(\ \perp\!\!\!\perp, a_2, ..., a_n) \leq^n (\ \perp\!\!\!\perp,\ \top\!\!\!\top, ...,\ \top\!\!\!\top)$$

Or la fonction $f^{\#}$ est monotone. Par conséquent, on peut écrire

$$f^{\#}(\ \perp\!\!\!\perp, a_2, ..., a_n) \leq f^{\#}(\ \perp\!\!\!\perp,\ \top\!\!\!\top, ...,\ \top\!\!\!\top).$$

On obtient donc l'implication suivante

$$(f^{\#}(\ \perp\!\!\!\perp,\ \top\!\!\!\top, ...,\ \top\!\!\!\top) = \ \perp\!\!\!\perp) \Rightarrow (f^{\#}(\ \perp\!\!\!\perp, a_2, ..., a_n) = \ \perp\!\!\!\perp).$$

Cette implication permet une définition de la "strictness" d'une fonction abstraite à partir d'un nombre restreint de points. Ainsi si une fonction f a deux arguments les seules valeurs de $f^{\#}$ qui nous intéressent pratiquement sont $f^{\#}(\ \bot,\ \top)$ et $f^{\#}(\ \top,\ \bot)$.

Cette définition restreinte est par ailleurs cohérente avec la signification de l'élément abstrait $\top$: "n'importe quel élément".

Le principe est donc de ne développer que les valeurs nécessaires. Qu'obtient-on dans le cas de nos deux exemples ?

Le premier programme est constitué de deux fonctions : une d'arité 1 et une autre d'arité 2. On a donc a priori 3 valeurs d'intérêt; c'est ce que traduit l'étape d'initialisation.

$$\mathcal{F}_0^{\#} : \begin{cases} f : \begin{array}{|c|c|} \hline (\ \bot,\ \top) & (\ \top,\ \bot) \\ \hline \bot & \bot \\ \hline \end{array} \\ g : \begin{array}{|c|} \hline \bot \\ \hline \bot \\ \hline \end{array} \end{cases}$$

On passe alors à la première itération. Au cours de l'évaluation de $(\mathcal{F}_1^{\#} f)(\ \top,\ \bot)$, on se rend compte qu'on a besoin de la valeur $(\mathcal{F}_0^{\#} g)(\ \top)$ qui n'existe pas dans la table. On ajoute alors la case correspondante, la valeur étant approximée par $\bot$.

$$\mathcal{F}_1^{\#} : \begin{cases} f : \begin{array}{|c|c|} \hline (\ \bot,\ \top) & (\ \top,\ \bot) \\ \hline \bot & \bot \\ \hline \end{array} \\ g : \begin{array}{|c|c|} \hline \bot & \top \\ \hline \bot & \bot \\ \hline \end{array} \end{cases}$$

La seconde itération donne les tables suivantes.

$$\mathcal{F}_2^{\#} : \begin{cases} f : \begin{array}{|c|c|} \hline (\ \bot,\ \top) & (\ \top,\ \bot) \\ \hline \bot & \top \\ \hline \end{array} \\ g : \begin{array}{|c|c|} \hline \bot & \top \\ \hline \bot & \top \\ \hline \end{array} \end{cases}$$

La troisième itération fournit la même table et par conséquent l'algorithme s'arrête.

Ici, le procédé n'est pas très intéressant puisqu'on retombe sur le même nombre d'évaluations. Mais considérons maintenant le second exemple.

a^4	$(\mathcal{F}_0^\# f)a^4$	$(\mathcal{F}_1^\# f)a^4$	$(\mathcal{F}_2^\# f)a^4$	$(\mathcal{F}_2^\# f)a^4$
($\top$, $\top$, $\top$, $\bot$)	$\bot$	$\bot$	$\bot$	$\bot$
($\top$, $\top$, $\bot$, $\top$)	$\bot$	$\bot$	$\bot$	$\bot$
($\top$, $\bot$, $\top$, $\top$)	$\bot$	$\top$	$\top$	$\top$
($\bot$, $\top$, $\top$, $\top$)	$\bot$	$\top$	$\top$	$\top$
($\bot$, $\bot$, $\top$, $\top$)	?	$\bot$	$\top$	$\top$
($\bot$, $\bot$, $\top$, $\bot$)	?	$\bot$	$\bot$	$\bot$

Les points d'interrogation représentent les situations où la valeur n'est pas encore prise en considération.

On passe ici de 28 évaluations à 13. L'intérêt de la deuxième méthode s'accentue avec l'accroissement du nombre de variables des fonctions (si une fonction a n arguments, on a seulement n points d'intérêts, au départ, contre 2^n élements pour l'espace).

Passons maintenant à la partie de l'exposé relative à l'implémentation CaML des analyseurs.

5.2 Fonctions auxiliaires

Comme on travaille avec des ensembles finis, les fonctions seront représentées par des listes de couples. Les trois fonctions qui suivent sont liées à cette représentation.

```
dom : (a'* b') list ---> a' list
im : a' ---> (a'* b') list ---> b'
change : 'a ---> 'b ---> (a'* b') list ---> (a'* b') list
```

La fonction `dom` fournit la liste des premières composantes d'une liste de couples c'est-à-dire que si la liste de couples `lc` représente une fonction, `dom lc` représente le domaine de cette fonction.

`im el liste` symbolise la seconde composante du premier couple de `liste` dont la première composante vaut `el`; s'il n'existe pas de couple de ce type, il s'agit d'un message d'erreur. Dans les cas où `liste` représente une fonction, `im el liste` représente l'image du point représenté par `el` par cette fonction.

`change a b liste` correspond à la liste `liste` dont le premier couple du type `(a,bb)` a été remplacé par `(a,b)`. Si `liste` représente une fonction f, `change a b liste` traduit $f[a/b]$.

La fonction suivante est un "classique" puisqu'il s'agit de la fonction qui fournit le nombre d'éléments d'une liste.

```
card : a' list ---> int
```

5.3 Implémentation du développement complet

5.3.1 Domaine et primitives

Le domaine abstrait est représenté tout simplement par deux constructeurs.

```
type abs = B | T;;
```

En reprenant le formalisme du chapitre 2, on écrit : $(\ \bot\!\!\!\bot)^c = \mathtt{B}$ et $(\ \top\!\!\!\top)^c = \mathtt{T}$.

On n'a effectivement que deux primitives à représenter puisque $succ^\#$ et $pred^\#$ équivalent toutes les deux à l'identité. On représente respectivement les primitives $egal^\#$ et $si^\#$ par les fonctions CaML `ega` et `sia` :

$$\forall a_1, a_2 \in A,\ \mathtt{ega}\ (a_1, a_2)^c = (egal^\#(a_1, a_2))^c$$

$$\forall a_1, a_2, a_3 \in A,\ \mathtt{sia}\ (a_1, a_2, a_3)^c = (sia^\#(a_1. a_2, a_3))^c$$

```
let ega = function
      (B,_) -> B
      | (_,B) -> B
      |(T,T) -> T;;

let sia = function
      (B,_,_) -> B
      |(T,B,B) -> B
      |(T,_,_) -> T;;
```

5.3.2 Environnements et fonction d'évaluation

Ce qui nous intéresse est de pouvoir décrire "en extension" la sémantique abstraite d'un programme. On représente donc chaque fonction comme une liste de couples du type `(abs list)*abs`. On obtient les types suivants pour les environnements :

```
type venva= VENVA of (string -> abs);;

type fenva = FENVA of (string*((abs list*abs) list)) list;;
```

$$(.)^c : I\!Env Var^{\#} \longrightarrow \texttt{venva}$$
$$(.)^c : I\!Env Fonc^{\#} \longrightarrow \texttt{fenva}$$

De nouveau, pratiquement on laisse souvent tomber les constructeurs.

On peut maintenant transposer la définition récursive de la fonction d'évaluation abstraite $\varepsilon^{\#}$ sur sa représentation CaML `evala`. De nouveau, on modifie l'ordre de la signature.

$$\forall \mathcal{F}^{\#} \in I\!Env Fonc^{\#},\ \forall \mathcal{V}^{\#} \in I\!Env Var^{\#},\ \forall expr \in I\!Expr,$$
$$\texttt{evala}(\mathcal{F}^{\#})^c(\mathcal{V}^{\#})^c expr^c = (\varepsilon^{\#}[\![expr]\!]\mathcal{F}^{\#}\mathcal{V}^{\#})^c$$

```
let rec evala (FENVA fe) (VENVA e) = function
      ZERO -> T
      |(VAR x)  -> e x
      |(SUIV expr) -> (evala (FENVA fe) (VENVA e) expr)
      |(PRED expr) -> (evala (FENVA fe) (VENVA e) expr)
      |(EGAL (ex1,ex2))
        -> ega (evala (FENVA fe) (VENVA e) ex1 ,
                evala (FENVA fe) (VENVA e) ex2)
      |(SI (ex1, ex2, ex3))
        -> sia (evala (FENVA fe) (VENVA e) ex1,
               evala (FENVA fe) (VENVA e) ex2,
               evala (FENVA fe) (VENVA e) ex3)
      |(FONC (s, l))
        ->  (im (appl (evala (FENVA fe) (VENVA e)) l) (im s fe));;
```

5.3.3 Sémantique abstraite d'un programme

5.3.3.1 Initialisation

Il nous faut, à partir de la représentation CaML d'un programme, construire la représentation CaML du premier élément de la suite de Kleene $\mathcal{F}_0^{\#}$. Cette représentation doit donc être du type `fenva`. Pour cela, on construit successivement les fonctions CaML

ajout, voulu, foncnulle et init qui vérifient :

$$\texttt{ajout}\ (A^i)^c = (A^{i+1})^c$$

$$\texttt{voulu}\ n^c = (A^n)^c$$

$$\texttt{foncnulle}\ (A^n)^c = (\lambda a^n : A^n.\ \bot\!\!\!\bot)^c$$

$$\texttt{init}\ (prog)^c = (\mathcal{F}_0^\#)^c$$

```
let rec ajout = function
      [] -> []
      | l::ll -> (T::l)::((B::l)::(ajout ll));;

let rec voulu = function
      0 -> [[]]
      | n -> ajout (voulu (n-1));;

let rec foncnulle =  function
      [] -> []
      | el::l -> (el,B)::(foncnulle l);;

let rec init = function
     (PROG [])  -> []
     | (PROG (((f,lv),expr)::p))
       -> ((f,foncnulle (voulu (card lv))))
          ::(init (PROG p));;
```

5.3.3.2 Itérations

Il s'agit simplement de traduire, pour tout programme p de l'ensemble $\mathbb{P}rog$, la transformation $\tau^\# [\![p]\!]$.

On définit d'abord les fonctions venvnulla et fonctionna qui vérifient

$$\texttt{venvnulla} = (\mathcal{V}_\bot^\#)^c$$

$$\forall expr \in \mathbb{E}xpr,\ \forall x_1, ..., x_n \in \mathbb{V}ar,\ \forall \mathcal{F}^\# \in \mathbb{E}nvFonc^\#, \forall f \in \mathbb{F}onc,$$
$$\texttt{fonctiona}\ (expr)^c (x_1, ..., x_n)^c (\mathcal{F}^\#)^c ((\mathcal{F}^\#) f)^c = (\lambda a^*.\varepsilon^\# [\![expr]\!] \mathcal{F}^\# \mathcal{V}_{a^*}^\#)^c.$$

```
let venvnula s = B;;
```

```
let rec fonctiona expr lv fe = function
      [] -> []
      | (a,fa)::l
       ->  if fa=T then
            (a,T)::(fonctiona expr lv fe l)
           else
            (a,evala (FENVA fe) (VENVA (ponctlist venvnula lv a)) expr)::
            (fonctiona expr lv fe l);;
```

Remarquons que si la fonction est déjà à la valeur `T`, on ne fait pas d'appel à la fonction `evala`.

Une fois la fonction `fonctiona` définie, la construction de la fonction CaML `transa` correspondant à la fonction sémantique $\tau^{\#}$ est la transposition directe de la définition de $\tau^{\#}$ selon la longueur du programme.

```
let rec transa = fun
      (PROG []) fe -> fe
      | (PROG (((f,lv),expr)::p)) fe
      -> change f (fonctiona expr lv fe (im f fe)) (transa (PROG p) fe);;
```

La fonction `transa` vérifie évidemment l'équation

$$\forall p \in I\!Prog,\ \forall \mathcal{F}^{\#} \in I\!EnvFonc^{\#},\ \texttt{transa}\ p^{c}(\mathcal{F}^{\#})^{c} = (\tau^{\#}[\![p]\!]\mathcal{F}^{\#})^{c}.$$

5.3.3.3 Point fixe

Il nous suffit maintenant de réappliquer la transformation $\tau^{\#}[\![p]\!]$ tant qu'elle produit un changement. Cette opération est réalisée par la fonction CaML `transform`:

```
transform : prog ---> fenva ---> fenva

let rec transform p fe = if (transa p fe = fe)
                         then fe
                         else (transform p (transa p fe));;
```

Cette fonction doit vérifier la proposition suivante.

$$\forall p \in I\!Prog,\ \texttt{transform}\,(\texttt{init}\ p^{c}) = (\mu^{\#}[\![p]\!])^{c}$$

5.4 Implémentation du développement restreint

On conserve évidemment les représentations du domaine abstrait et des primitives abstraites présentées précédemment ainsi que les représentations des ensembles $\mathbb{E}nvVar^{\#}$ et $\mathbb{E}nvFonc^{\#}$. Cependant, comme on ne cherche plus la sémantique abstraite d'un programme dans son entièreté, on ne travaille plus réellement avec des environnements de fonctions complets mais avec les "morceaux utiles" de ceux-ci. On ne peut donc plus utiliser le formalisme employé jusqu'à présent pour les spécifications. Par conséquent, on décrira les fonctions CaML utilisées de manière plus informelle.

5.4.1 Fonction d'évaluation

En cours d'évaluation, il se peut qu'on se rende compte qu'on a besoin d'une valeur supplémentaire (i.e. d'une valeur d'une fonction en un point non repris dans l'environnement restreint courant). Dans ce cas, on ajoute un élément à l'environnement de fonctions. L'environnement de fonctions peut donc être modifié par l'évaluation d'une expression. C'est pourquoi, on modifie la signature de la fonction `evala`.

```
evala : fenva ---> venva ---> expr ---> abs*fenva
```

A chaque appel de fonction, il faut tester si le point correspondant appartient déjà à l'environnement courant. Si ce n'est pas le cas, il faut rajouter un élément à l'environnement. Expliquons cela de manière un peu plus précise.

Soit un appel de fonction `(f,lexpr)`. On évalue la liste d'expressions et on obtient normalement une liste de valeurs abstraites correspondantes `lval`. On regarde alors si le domaine de la liste associée à f dans l'environnement courant contient `lval`. Dans les cas positifs, on renvoie juste la valeur fournie (l'environnement restant inchangé). Dans les cas négatifs, on renvoie `B` et on ajoute à la fin de la liste associée à f la cellule `(lval, B)`.

Une autre modification résulte du besoin de "propager" les changements d'environnements: par exemple, on introduit la fonction `evallist` pour propager d'éventuels changements d'environnements lors de l'évaluation de la liste des arguments d'un appel de fonction (l'évaluation d'un argument pouvant en fait lui-même engendrer une modification de l'environnement de fonction). Il s'agit en fait tout simplement de la prise en compte de la deuxième composante du résulat de la fonction.

```
let rec evala (FENVA fe) (VENVA e) = function
      ZERO -> (T,fe)
      |(VAR x)  -> (e x,fe)
      |(SUIV expr) -> (evala (FENVA fe) (VENVA e) expr)
      |(PRED expr) -> (evala (FENVA fe) (VENVA e) expr)
      |(EGAL (ex1,ex2))
        -> let (v1,fe1) = evala (FENVA fe) (VENVA e) ex1
           in let (v2,fe2) = evala (FENVA fe1) (VENVA e) ex2
           in  (ega (v1,v2),fe2)
      |(SI (ex1, ex2, ex3))
        -> let (v1,fe1) = evala (FENVA fe) (VENVA e) ex1 in
           if v1 = B then
             (B,fe1)
           else let (v2,fe2)= evala (FENVA fe1) (VENVA e) ex2
                in let (v3,fe3) = evala (FENVA fe2) (VENVA e) ex3
                in ((sia (T,v2,v3)),fe3)
      |(FONC (s, l))
        ->  let (lv,fe2) = evallist (FENVA fe) (VENVA e) l
            in
            let ff = im s fe2
            in
            if dans lv (dom ff)
            then (im lv ff, fe2)
            else  (B,change s (ff@[(lv,B)]) fe2)
and
      evallist (FENVA fe) (VENVA e) = function
            [] -> ([],fe)
            | el::l -> let (l1,fe1) = evallist (FENVA fe) (VENVA e) l
                       in let (v1,fe2) = evala (FENVA fe1) (VENVA e) el
                       in (v1::l1,fe2);;
```

5.4.2 Sémantique restreinte d'un programme

5.4.2.1 Initialisation

Il ne s'agit plus maintenant de développer pour chaque fonction l'espace complet A^n correspondant à son arité n, mais juste les points de cet espace dont une et une seule composante vaut $\perp\!\!\!\perp$. C'est ce que réalise la fonction `voulu : int ---> (abs list) list` dont le code est fourni en annexe. A partir de là, on peut reprendre exactement le code de la fonction `init` présentée pour le développement complet.

5.4.2.2 Itérations

L'éventuelle modification de l'environnement par la fonction d'évaluation a une autre conséquence: le développement d'une fonction peut modifier l'environnement de fonctions pour les autres fonctions. On ne peut donc plus avoir de fonction CaML comme `fonctionna` qui renvoie uniquement la liste représentant la fonction sémantique correspondant à un symbole fonctionnel particulier, il faut maintenant renvoyer un environnement de fonctions dans son entièreté. C'est ce que réalise la fonction `devfonc`, la fonction `transa` est légèrement modifiée en conséquence.

```
devfonc : string ---> (string list) ---> expr ---> fenva
          ---> (abs list*abs) list ---> fenva
transa : prog ---> fenva ---> fenva

let rec devfonc f lv expr fe = function
      [] -> fe
      | (a,fa)::l
      -> if fa=T
         then (devfonc f lv expr fe l)
         else
          let (v1,fe1)
           = evala (FENVA fe) (VENVA (ponctlist venvnula lv a)) expr
           in
           devfonc f lv expr (change f (change a v1 (im f fe1)) fe1) l;;

let rec transa = fun

      (PROG []) fe -> fe
      | (PROG (((f,lv),expr)::ldec)) fe
        -> let fe2 = transa (PROG ldec) fe
           in devfonc f lv expr fe2 (im f fe2);;
```

Remarquons qu'on a implicitement introduit un autre changement algorithmique.

Dans les algorithmes décrits, on effectue les évaluations de l'itération $(i+1)$ dans l'environnement de fonctions $\mathcal{F}_i^{\#}$. Ce n'est pas cette idée qu'on utilise ici puisqu'on réutilise certaines "valeurs plus récentes". On travaille en fait dans un environnement de fonctions courant $(\mathcal{F}_i^{\#})'$ qui vérifie

$$\mathcal{F}_i^{\#} \leq^{ef} (\mathcal{F}_i^{\#})' \leq^{ef} \mathcal{F}_{i+1}^{\#}.$$

Remarquons que cette inégalité du type "étau", permet de garantir qu'on conserve bien la valeur du plus petit point fixe.

La fonction `transform` relative au point fixe est inchangée.

5.5 Modification de l'interpréteur

Pour exploiter l'information obtenue par l'analyseur, on procède comme suit : on extrait d'abord les informations utiles du modèle abstrait, on modifie ensuite le programme en ajoutant à chaque appel de fonction un indicateur du type de passage d'argument à effectuer et, pour terminer, on applique au programme un interpréteur disposant d'une fonction d'évaluation pouvant réaliser le passage par valeur comme le passage par nom.

On pourrait envisager une autre solution qui consisterait à conserver dans l'environnement les informations nécessaires.

5.5.1 Extraction des informations

Dans le cas du développement complet, l'analyseur fournit la table complète de la sémantique abstraite. Or les valeurs des fonctions ne nous intéressent qu'en certains points (une et une seule composante à $\perp\!\!\!\perp$). L'épuration est réalisée par la fonction `epure2` : `fenva ---> fenva`, elle garantit en outre le séquencement des valeurs selon la position du $\perp\!\!\!\perp$ dans le vecteur. Le code de cette fonction est fourni en annexe.

Le cas du développement restreint est encore plus simple : les points inutiles sont systématiquement placés en fin de listes par la fonction `evala`; par conséquent, pour une fonction d'arité n, il suffit de considérer seulement les n premières valeurs de la liste associée à la fonction.

Quoi qu'il en soit, dans les deux cas, on se retrouve avec un élement de type `fenva` contenant toute l'information nécessaire et "correctement" ordonnée.

5.5.2 Modification du programme

L'idée est donc d'ajouter un indicateur du type de passage d'argument à chaque appel de fonction.

```
type tparam = TVAL | TNOM;;
```

On définit donc un type CaML correspondant aux expressions modifiées. Il ne diffère du type `expr` que par les constructeurs et l'appel de fonction.

```
type exprmod = Z | V of string | S of exprmod | P of exprmod
             | EG of (exprmod*exprmod)
             | SII of (exprmod*exprmod*exprmod)
             | F of string*(exprmod*tparam) list;;
```

On construit alors successivement les fonctions `modif`, `modifexpr` et `modifprog`.

```
modif : exprmod list ---> (abs list * abs) list
       ---> (exprmod*tparam) list
modifexpr : fenva ---> expr ---> exprmod
modifprog : fenva ---> prog
           ---> ((string*string list)*exprmod) list
```

La fonction `modif` ajoute le "filtre" des indicateurs à la liste d'expressions modifiées correspondant aux arguments d'un appel de fonctions.

```
let rec modif = fun
      [] l -> []
      | (e::le) ((la,a)::ll)
       -> if a=B then ((e,TVAL)::(modif le ll))
                 else ((e,TNOM)::(modif le ll));;
```

La fonction `modifexpr` modifie une expression au vu d'un élément de type `fenva`. Tandis que la fonction `modifprog` réalise la même opération pour un programme.

```
let rec modifexpr te = function
      ZERO -> Z
      | VAR x -> V x
      | SUIV expr -> S (modifexpr te expr)
      | PRED expr -> P (modifexpr te expr)
      | EGAL (e1,e2) -> EG (modifexpr te e1, modifexpr te e2)
      | SI (e1,e2,e3) -> SII (modifexpr te e1, modifexpr te e2,
                              modifexpr te e3)
      | FONC (s,le) -> F (s,modif (appl (modifexpr te) le) (im s te));;

let rec modifprog te = function
      (PROG []) -> []
      | (PROG (((f,lv),expr)::ldec))
      -> ((f,lv),modifexpr te expr)::(modifprog te (PROG ldec));;
```

5.5.3 Interpréteur

Un appel de fonction modifié contient un indicateur du type de passage d'argument à effectuer et ce pour chaque argument. Le principe est donc de construire une fonction d'évaluation qui utilise cet indicateur pour décider si elle doit ou pas envelopper l'argument en question dans une fonction.

La seule fonction de l'interpréteur qui est modifiée est donc de nouveau la fonction d'évaluation.

Il nous faut définir de nouveaux types pour les environnements, permettant de traiter les arguments des fonctions tantôt comme des valeurs simples tantôt comme des fonctions constantes.

```
type param = VAL of val | NOM of (int->val);;

type venv = VENV of (string -> param);;

type fenv = FENV of string->((param list) -> val);;
```

La fonction `eval` travaille maintenant avec des expressions modifiées.
Lors de l'évaluation d'un appel de fonctions, la fonction `enveloppe` traite la liste des arguments et des indicateurs : si un argument est associé à l'indicateur `TNOM`, elle l'enveloppe sinon elle le laisse tel quel.

```
let rec eval (FENV fe) (VENV e) = function
      Z -> (NAT 0)
      |(V x)  -> (match e x with
                    VAL v -> v
                    | NOM f -> f 0)
      |(S expr) -> succ(eval (FENV fe) (VENV e)  expr)
      |(P expr) -> pred(eval (FENV fe) (VENV e)  expr)
      |(EG (ex1,ex2))
        -> eg (eval (FENV fe) (VENV e)  ex1 ,
               eval (FENV fe) (VENV e)  ex2)
      |(SII (ex1, ex2, ex3))
        -> if (eval (FENV fe) (VENV e)  ex1) = BOT
           then BOT
           else
             begin
               if (eval (FENV fe) (VENV e)  ex1) = (NAT 1)
```

```
                then eval (FENV fe) (VENV e) ex2
                else eval (FENV fe) (VENV e) ex3
              end
      |(F (s, l))
        -> fe(s) (enveloppe (FENV fe) (VENV e)  l)
and
      envel fe e expr x = eval fe e expr

and
      enveloppe fe e  = function
            [] -> []
            | ((expr,TVAL)::le)
             -> (VAL (eval fe e expr))::(enveloppe fe e le)
            | ((expr,TNOM)::le)
             -> (NOM (envel fe e expr))::(enveloppe fe e le);;
```

On modifie également légèrement la fonction d'évaluation de telle manière qu'elle lance l'analyseur et modifie le programme et les expressions à évaluer en fonction.

```
let evaluation (EXEC (p,e)) =

    if verifexec (EXEC (p,e))
    then let pp = (transform (init p)) in
          appl (eval (FENV (modele (modifprog pp p))) (VENV venvnul))
          (appl (modifexpr pp) e)
    else raise (Erreur "Execution incorrecte");;
```

5.6 Tests

Pour clôturer ce chapitre, on réalise quelques tests. Dans un premier temps. on compare les deux analyseurs et dans un second temps, on compare les performances du nouvel interpréteur et de l'interpréteur utilisant le passage par valeur.

5.6.1 Comparaison des deux analyseurs

Désignons l'analyseur en développement complet par l'abréviation $an1$ et l'analyseur en développement restreint par l'abréviation $an2$. On considère de nouveau les programmes p_1 et p_2 présentés à la fin du chapitre 2 (cfr pages 53 et 55). On considère

également de nouveaux programmes $p_{1(i)}$ qui ne diffèrent du programme p_1 que par le nombre d'arguments de la fonction *grand*: dans le programme $p_{1(i)}$, la fonction *grand* est d'arité i et calcule la somme de ses i arguments.

Le tableau suivant reprend les temps mis par les deux analyseurs pour traiter le programme (exprimés en dixièmes de secondes).

	$an1$	$an2$
p_2	6	6
$p_{1(4)}$	8	6
p_1	18	7
$p_{1(8)}$	84	7
$p_{1(10)}$	–	9

Sur ces exemples, il est assez évident que le second algorithme est préférable au premier. Remarquons que pour le programme p_1, l'analyseur détecte que toutes les fonctions sont strictes et que pour le programme p_2, il détecte que toutes les fonctions sont strictes sauf la fonction *fois* en sa seconde variable. On a donc, pour ces programmes, les résultats attendus.

On peut produire un tableau analogue à celui ci-dessus mais comparant cette fois les "tailles" des environnements (on donne en fait, comme dans les exemples de la première partie du chapitre, le nombre d'évaluations nécessaires).

	$an1$	$an2$
p_2	$14 + 9 = 23$	$7 + 7 = 14$
$p_{1(4)}$	$16 + 15 + 14 = 45$	$6 + 7 = 13$
p_1	$68 + 67 + 66 = 201$	$8 + 9 = 17$
$p_{1(8)}$	$260 + 259 + 258 = 777$	$10 + 11 = 21$
$p_{1(10)}$	$1028 + 1027 + 1026 = 3081$	$12 + 13 = 25$

Les termes des sommes correspondent aux itérations du point fixe[2]. On remarque que dans les exemples du type $p_{1(i)}$, l'effet de la première petite optimisation est quasiment négligeable.

5.6.2 Comparaison des interpréteurs

Dans cette section, on ne considère plus que l'analyseur résultant du second algorithme (i.e. l'analyseur en développement restreint). Les temps d'analyse sont la plupart

[2]En extrapolant, on peut se dire que pour $p_{1(i)}$, on obtiendrait pour $an1$ un calcul du type $3*(2^i+2^2)-3$ et pour $an2$ un calcul du type $2*(i+2)+1$.

du temps négligeables. On compare maintenant les temps d'éxécution des différents interpréteurs. On symbolise le nouvel interpréteur "hybride" par *hyb*.

On considère de nouveau les programmes p_1 et p_2. Le tableau ci-dessous fournit les temps d'exécution (en secondes) pour des expressions à evaluer du type

$$grand(expr, expr, expr, expr, expr, expr).$$

"$\varepsilon[\![expr]\!]$"	*nom*	*valeur*	*hyb*
1	0	0	0
2	2	0	0
3	6	0	0
4	12	0	0
5	24	0	0
6	42	0	0
7	69	0	0
8	–	0	0
50	–	1	1
100	–	2	2
200	–	4	5
1000	–	24	25
2000	–	51	54
2200	–	57	61
2300	–	61	–
2400	–	–	–

Dans ce cas, les fonctions sont toutes strictes et les performances sont quasiment identiques.

Considérons maintenant, des exemples d'évaluations liées à p_2. On envisage successivement des expressions du type $fact(expr)$ et $fois(expr, expr)$.

"$\varepsilon[\![expr]\!]$"	*nom*	*valeur*	*hyb*
2	0	0	0
3	4	0	0
4	–	0	1
5	–	0	4
6	–	2	22
7	–	13	–
8	–	–	–

"$\varepsilon[\![expr]\!]$"	*nom*	*valeur*	*hyb*
10	1	0	0
20	5	1	1
30	17	2	2
40	38	3	4
50	–	5	6
100	–	21	24
150	–	50	56
170	–	64	72
180	–	72	81
190	–	81	–
200	–	–	–

Soulignons que dans ce cas, il existe une fonction pour laquelle un des arguments est passé par nom. Il est donc normal d'avoir des temps d'exécution plus importants.

5.6.3 Un dernier exemple

Pour terminer notre exposé, considérons un programme "un peu plus tordu", où les fonctions s'enchevêtrent plus ...

$$p_3 : \begin{cases} f(x,y,z) = & \textbf{si } x = 0 \\ & \textbf{alors } \textbf{si } y = 0 \\ & \qquad \textbf{alors } 1 \\ & \qquad \textbf{sinon } f(x, y-1, h(f(x,y,z), g(x,y,z).x)) \\ & \textbf{sinon } g(x-1,y,z) \\ g(x,y,z) = & f(x,y,z) \\ h(x,y,z) = & \textbf{si } x = 0 \\ & \textbf{alors } f(x,y,z) \\ & \textbf{sinon } g(x,y,z) \end{cases}$$

Intuitivement, les trois fonctions de ce programme sont absolument identiques. Leur sémantique est égale à

$$\lambda(x,y,z).(\text{si } (x = \bot) \vee (y = \bot) \text{ alors } \bot \text{ sinon } 1).$$

On a que les analyseurs détectent bien que les trois fonctions sont strictes en les deux premières variables. L'interpréteur hybride passera donc les paramètres correspondants par valeur mais passera les troisièmes arguments par nom.

Le premier analyseur demande 1 seconde pour traiter le programme p_3 tandis que le second n'en réclame que 0,6. On a donc, dans ce cas, que malgré l'enchevêtrement des fonctions, ce dernier reste plus intéressant.

En guise de conclusion

Si on considère que notre objectif était de développer entièrement un problème par l'interprétation abstraite, on peut prétendre l'avoir atteint.

En effet, après avoir trouvé un formalisme mathématique adéquat pour exprimer la notion de fonction de stricte, on a pu utiliser cette notion pour réaliser une analyse de programmes d'un langage applicatif dans le domaine abstrait $\{ \perp\!\!\!\perp, \top\!\!\!\top \}$. Les théorèmes du chapitre 3 assurent en outre la correction de notre démarche.

De plus, on a trouvé le moyen d'utiliser les résultats obtenus pour générer un interpréteur "hybride" conservant la définition "naturelle" du passage par nom des arguments mais nettement plus efficace qu'un interpréteur fonctionnant uniquement en passage par nom.

Il est cependant évident que ce travail n'est ni parfait ni "complet". Par exemple, qu'entend-on exactement par "nettement plus efficace" dans le paragraphe ci-dessus? Peut-on se contenter des quelques chiffres fournis? ...

Nous mentionnerons ici trois grandes "carences" de l'exposé.

- Nous avons, au cours du chapitre 4, signalé l'incomplétude de notre analyse en corroborant notre propos par deux exemples. S'il est évident, au vu des résultats d'indécidabilité de la théorie de la calculabilité, que l'obtention d'une démarche complète est impossible au sens évoqué dans ce chapitre, un problème important serait certainement la mesure de la "puissance" de l'analyse.

  Pour ce faire, certains auteurs introduisent une autre notion de complétude souvent désignée par le terme d'optimalité. Il se trouve qu'on peut démontrer, dans ce sens, que notre analyse de "strictness" est complète (cfr [10]).

- Lors du dernier chapitre, nous avons présenté le deuxième analyseur ainsi que la construction de l'interpréteur hybride de manière tout à fait informelle. Nous en avons simplement expliqué les principes de fonctionnement sans réellement spécifier celui-ci. Sans spécification, il nous est évidemment impossible d'exprimer correctement que notre algorithme "fait bien ce que l'on veut". Il serait donc opportun

de trouver un formalisme pouvant traduire, par exemple, notre concept d'environnement de fonctions restreint. On trouvera cependant une démonstration, dans un cadre plus général, de la correction de cet algorithme dans [6].

Toujours en ce qui concerne la présentation des algorithmes, il est assez évident qu'on pourrait réaliser une étude de complexité de ceux-ci permettant notamment une comparaison plus générale des performances de l'un et de l'autre. Il se trouve que dans "le pire des cas" les deux algorithmes sont équivalents. Cependant, une étude plus fouillée apporterait certainement une meilleure compréhension du nombre d'itérations nécessaires.

- Le dernier point réside dans la généralisation de l'analyse. Que se passe-t-il quand on passe aux langages fonctionnels de "plus haut niveau" ? Rappelons qu'on parle de langages fonctionnels d'ordre supérieur quand les fonctions peuvent renvoyer comme résultats des objets composés comme des vecteurs ou des fonctions. L'analyse employée est-elle toujours correcte? Les algorithmes sont-ils toujours valables? Et si oui, sont-ils toujours suffisamment efficaces? On se rend déjà compte que conserver toute la "table" des valeurs risque de poser certains problèmes et ce même pour un domaine de base à deux valeurs ...

  Pour répondre à ces questions, il nous faudrait reprendre toute l'analyse effectuée dans ce travail en considérant, par exemple, un langage du type "lambda calcul typé" (cfr [10] et [2]).

Pour terminer ce travail, mentionnons encore un problème au niveau de la forme. Nous avons employé, dans notre propos, des dénominations abusives et souvent désigné des objets de types différents par des locutions identiques. Il est par exemple assez difficile de trouver des dénominations différentes pour les nombreuses significations du mot "fonction".

Il serait peut-être opportun de parvenir à distinguer au niveau des dénominations des objets provenant de "mondes différents". On pourrait sans doute y parvenir par un système de préindexation de certains termes. L'inconvénient majeur de ce genre de dénominations est d'alourdir fortement le propos.

Bibliographie

1. S. Abramsky and C. Hankin, eds. *Abstract Interpretation of Declarative Languages.* Computers and their Applications. Ellis Horwood, 1987.

2. P. Cousot and R. Cousot. Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages). In *Proccedings of the 1994 International Conference on Computer Languages* (ICCL'94), pages 95-112, Toulouse, France, May 16-19 1994. IEEE Press, Los Alamitos, California. (Invited paper).

3. B. Le Charlier. *Computational Logic.* Notes de cours, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur.

4. B. Le Charlier. L'analyse statique des programmes par l'interprétation abstraite. Dans *Nouvelles de la Science et des Technologies*, vol. 9, numéro 4, 1991, pp 19-26.

5. B. Le Charlier and P. Flener. On the Desirable Link Between Theory and Practice in Abstract Interpretation. In *Proceedings of SAS'97*, September 1997. (Extended Abstract).

6. B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report 92-22, Institute of Computer Science, University of Namur, Belgium, April 1992.

7. X. Leroy. The CaML Light System, Release 0.6: Documentation and User's Manual. Technical Report, INRIA, 1993.

8. C. Livercy. *Théorie des programmes : schémas, preuves, sémantique.* Dunod, Paris. 1978.

9. A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicatives Programs.* PhD thesis, University of Edinburgh, Dec. 1981.

10. U.S. Reddy and S.N. Kamin. On the Power of Abstract Interpretation. In J. Cordy, editor. *Proceedings of the IEEE fourth International Conference on Computer Language* (ICCL'92). U.S.A., April 1992. IEEE Press.

11. J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Mass., 1977.

Annexes

Fonctions auxiliaires

Cette section fournit simplement le code des fonctions auxiliaires mentionnées lors de la présentation des implémentations au cours des chapitres 2 et 5. On adjoint à celles-ci quelques autres fonctions relatives à la traduction des ensembles par des listes.

```
let rec dans e = function
      [] -> false
      | ee :: l -> (ee=e) or (dans e l);;

let rec ens = function
      [] -> []
      | e::l -> if dans e l then (ens l) else e::(ens l);;

let rec estens = function
      [] -> true
      | e::l -> (not (dans e l)) & (estens l);;

let rec contient b = function
      [] -> true
      | e::a -> (dans e b) & (contient b a);;

let rec inter b = function
      [] -> []
      | e::a -> if (dans e b) then (e::(inter b a)) else (inter b a);;

let rec card = function
      [] -> 0
      | el::l -> (card l) + 1;;

let rec im a = function
      [] -> raise (Erreur "element pas dans la liste")
      | (aa,b)::l -> if a=aa then b else (im a l);;
```

```
let rec dom = function
      [] -> []
      |(a,b)::l ->   a::(dom l);;

let rec change a b = function
      [] -> []
      | (aa,bb)::l -> if a=aa then (a,b)::l else (aa,bb)::(change a b l);;

let rec appl f = function
      [] -> []
      | el::l -> f(el) :: (appl f l);;

let ponct f a b x = if (x =a) then b else f x ;;

let rec ponctlist f = fun
      [] [] -> f
      | (a::lista) (b::listb)
        -> let ff=ponctlist f lista listb in ponct ff a b;;
```

Fonctions de vérification syntaxique

On fournit ici le code des fonctions intervenant dans la construction de la fonction `verifexec` qui permet la vérification syntaxique de la représentation CaML d'une exécution.

```
let rec vars = function
      ZERO -> []
      | VAR x -> [x]
      | SUIV e -> vars e
      | PRED e -> vars e
      | EGAL (e1, e2) -> ens ((vars e1)@(vars e2))
      | SI (e1,e2,e3) -> ens (((vars e1)@(vars e2))@(vars e3))
      | FONC (f,[]) -> []
      | FONC (f,e::l) -> ens ((vars e)@vars (FONC (f,l)));;

let rec foncts = function
      ZERO -> []
      | VAR x -> []
      | SUIV e -> foncts e
      | PRED e -> foncts e
      | EGAL (e1, e2) -> ens ((foncts e1)@(foncts e2))
      | SI (e1,e2,e3) -> ens (((foncts e1)@(foncts e2))@(foncts e3))
```

```
      | FONC (f,[]) -> [f]
      | FONC (f,e::l) -> ens ((foncts e)@foncts (FONC (f,l)));;

let rec fonctdecl = function
      PROG [] -> []
      | PROG (((s,l),e) :: ldec) -> (s::(fonctdecl (PROG ldec)));;

let rec verifdecl  = function
     PROG [] -> true
     | PROG (((s,l),e)::p)
       -> if (contient l (vars(e)))
          then verifdecl (PROG p)
          else
           raise (Erreur ("Variable inconnue dans la declaration de "^s));;

let rec varsprog = function
      PROG [] -> []
     | PROG (((s,l),e)::p) -> ens(vars (e) @ varsprog (PROG p));;

let rec fonctsprog = function
      PROG [] -> []
     | PROG (((s,l),e)::p) -> ens(foncts (e) @ fonctsprog (PROG p));;

let rec aritedecl s = function
      (PROG []) -> raise  (Erreur ("Fonction non declaree : " ^ s))
      | PROG (((ss,l),e)::ldec)
        -> if  ss=s then (card l)
           else aritedecl s (PROG  ldec);;

let rec verifarite p = function
      ZERO -> true
      | VAR x -> true
      | SUIV (expr) -> verifarite p expr
      | PRED (expr) -> verifarite p expr
      | EGAL (ex1,ex2) -> (verifarite p ex1) & (verifarite p ex2)
      | SI (ex1,ex2,ex3) -> ((verifarite p ex1) & (verifarite p ex2)) &
                            (verifarite p ex3)

      | FONC (s,l) -> ((aritedecl s p)= (card l)) & (verifas p l)

and

      verifas p = function
            [] -> true
            | el::l -> (verifarite p el) & (verifas p l);;
```

```
let rec listexpr = function
      PROG [] -> []
      | PROG (((s,l),expr)::p) -> expr::(listexpr (PROG p));;

let verifR2 p = if verifas p (listexpr p) then
                (if (inter (varsprog p) (fonctsprog p)) = []
                then (if estens(fonctdecl p)
                     then (if contient (fonctdecl p) (fonctsprog p)
                           then verifdecl p
                           else raise
                            (Erreur "Utilisation d'une fonction
                                     non declaree dans la definiton
                                     du programme" ) )
                     else raise
                      (Erreur "Fonction declaree plus d'une fois") )
               else raise
                (Erreur "Confusion nom de fonction/nom de variable"))
               else raise
                (Erreur "Arite non respectee dans la definition
                                 du programme");;


let rec verifexec = function
      (EXEC (p,[])) -> verifR2 p
      | (EXEC (p,e::l))
        ->if (contient (fonctdecl p) (foncts e))
          then (if (vars e = [])
                then verifexec (EXEC (p,l))
                else raise
                  (Erreur "Presence d'une variable
                           dans les expressions à évaluer"))
          else raise (Erreur "Fonction non declaree");;

let verif (EXEC (p,l)) =
       if (verifas p l)
       then (verifexec (EXEC (p,l)))
       else raise (Erreur "Arite non respectee
                           dans les expressions a evaluer");;
```

Autres

Les quelques fonctions CaML ci-dessous permettent de construire les points d'intérêt, et par conséquent l'étape d'initialisation, de l'algorithme en développement restreint.

```
let rec ajout el = function
      [] -> []
      | l::ll -> (el::l)::(ajout el ll);;

let rec placer el = function
      [] -> [[el]]
      | a::l -> (el::(a::l))::(ajout a (placer el l));;

let rec ntop = function
      0 -> []
      | i -> T::(ntop (i-1));;

let rec voulu = function
      0 -> []
      | i -> placer B (ntop (i-1));;

let rec foncnulle =  function
      [] -> []
      | el::l -> (el,B)::(foncnulle l);;


let rec init = function
     (PROG [])  -> []
     | (PROG (((f,lv),expr)::p))
       -> ((f,foncnulle (voulu (card lv))))
          ::(init (PROG p));;
```

Les quelques fonctions ci-dessous correspondent à la phase d'extraction des résultats du modèle abstrait dans le cadre du développement complet.

```
let rec nbrebot = function
      [] -> 0
      | B::la -> nbrebot(la)+1
      |T::la -> nbrebot(la);;

let rec epure1 = function
      [] -> []
      | ((la,a)::ll) -> if nbrebot(la)=1 then (la,a)::(epure1 ll)
```

```
                        else (epure1 ll);;

let rec epure2 = function
      [] -> []
      | (s,l)::ls -> (s,epure1 l)::(epure2 ls);;

let modela p = epure2(transform p (init p));;
```